

AFRL-IF-RS-TR-2003-169
Final Technical Report
July 2003



A TEMPLATE-BASED PLANNING ASSOCIATE FOR ACTIVE TEMPLATES

ISX Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J764


APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

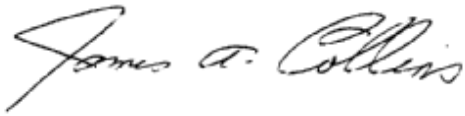
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-169 has been reviewed and is approved for publication.

APPROVED: 
DALE W. RICHARDS
Project Engineer

FOR THE DIRECTOR: 
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE JULY 2003	3. REPORT TYPE AND DATES COVERED Final May 00 – May 03	
4. TITLE AND SUBTITLE A TEMPLATE-BASED PLANNING ASSOCIATE FOR ACTIVE TEMPLATES			5. FUNDING NUMBERS C - F30602-00-C-0052 PE - 63706E PR - ATEM TA - PO WU - 08	
6. AUTHOR(S) Mark Hoffman and David Van Brackle				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ISX Corporation 760 Paseo Camarillo Suite 401 Camarillo California 93010			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/ITB 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-169	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Dale W. Richards/ITB/(315) 330-3014/ Dale.Richards@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) The Active Templates program developed innovative planning tools for the Special Operations Forces community. These software tools were developed by several contractors. This report documents technology developed to help integrate those tools. Initially, a simple tool was developed for managing plans based on Plan and Goal Graphs. Applications could connect to this tool via sockets. Although the tool had a good set of functionality and was well received, it was clearly inadequate for the larger program. Using this tool as a base, a more sophisticated socket-based XML message passing tool, the Router, and a common data repository, the Plan Manager were developed. The two together constituted the Tool Interchange Manager (TIM). External to this effort, a set of database tables describing concepts shared among the tools was developed - the Structured Data Model (SDM). This effort was then tasked with maintaining the SDM. After off-the-shelf infrastructure was found to meet the infrastructure needs of the program, work was stopped on the TIM and efforts directed at creating an application to allow a higher-level commander to coordinate the finer-grained plans developed by the various Active Templates tools - the Plan Coordinator.				
14. SUBJECT TERMS Active Template, Templated-Based Planning, Tool Interchange Manager, Plan Coordinator, Structured Data Model, Message-Oriented Middleware			15. NUMBER OF PAGES 66	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

TABLE OF CONTENTS

1	EXECUTIVE SUMMARY.....	1
2	ROUTER	1
3	TOOL INTERCHANGE MANAGER (TIM)	3
4	STRUCTURED DATA MODEL.....	7
5	PLAN COORDINATOR.....	14
6	LESSONS LEARNED	15
	APPENDIX A – JUMPSTART	16
	APPENDIX B TIM DEVELOPERS REFERENCE GUIDE	32
	APPENDIX C - ROUTER.....	47
	APPENDIX D – PLAN COORDINATOR USER’S GUIDE.....	50

LIST OF FIGURES

Figure 1 – Tool Interchange Manager (TIM)	4
Figure 2 – Application Layer (AL).....	5
Figure 3 – Work Center	6
Figure 4 – Work Center Window.....	7
Figure 5 – Initial SDM Plan Elements and Related Tables	9
Figure 6 - Constraint Tables w/o links to rest of SDM.....	11
Figure 7 - Resource Tables (without links to rest of SDM).....	12
Figure 8 - Simple SDM with implicit foreign relations.....	13
Figure 9 – Plan Coordinator.....	15

1 Executive Summary

The Active Templates program developed innovative planning tools for the Special Operations Forces community. These software tools were developed by several contractors. This report documents technology developed to help integrate those tools and facilitate their interoperation.

Initially, a simple tool was developed for managing plans based on Plan and Goal Graphs. Applications could connect to this tool via sockets. Although the tool had a good set of functionality and was well received, it was clearly inadequate for the larger program. Using this tool as a base, a more sophisticated socket-based XML message passing tool, the Router, and a common data repository, the Plan Manager were developed. The two together constituted the Tool Interchange Manager (TIM).

External to this effort, a set of database tables describing concepts shared among the tools was developed - the Structured Data Model (SDM). This effort was then tasked with maintaining the SDM.

After off-the-shelf technology was found to meet the infrastructure needs of the program, work was stopped on the TIM and efforts directed at creating an application to allow a higher-level commander to coordinate finer-grained plans developed by the various Active Templates tools - the Plan Coordinator

2 Router

The development of the Router began with an original Active Templates XML-Based Messaging Protocol (AXMP) specification. This then led to the MessageRouter, EchoMessageServer and Lax2MessageClient (originally just called MessageClient; the Lax2 was refactored out into a subclass later on). All of this was in response to a need to facilitate peer-to-peer like communications among components of the Active Templates system. Special emphasis was given to interoperation with the Plan Manager, an early Active Templates product which used XML as its primary language. Originally, the Plan Manager continued to use standard input and output streams to communicate with the Server implementation, which was handled by the MessageServer implementation through the ProcessStreamsMonitor, a JavaBean-like wrapper around the *java.lang.Process* class and its streams.

When the Plan Manager was replaced by the larger ContextManager, the communication direction was reversed. Instead of the Server implementation starting the manager, the manager started the router and server, then connected to it via sockets. The new Server implementation was ContextMessageServerSocket (CMSS). To conserve resources, the Router, CMSS, and other components making up the TIM (WorkgroupManager, Query Mediator, and WebScraper) were all run within the same Java Virtual Machine (JVM), through a utility called ShellRun, which would read a file of java classnames and launch each one's main() method in its own thread. At this time, the debugging interface, MessageRouterDebug, was standardized across most of the components.

The issue of encapsulating machine-to-machine communications came next in the evolution of the Router. The Router, being network-socket based, could easily support collaboration and communication on multiple machines talking to the same central router, but there was concern that these networks could be unreliable or be restricted by encryption or port and protocol-restricted firewalls.

The first version of router-to-router communication was the remote router concept. Here, a component inside the local router acted as a client to the remote router, and AXMP protocol 1.1 added the "remote" attribute to the core request and message tags. Here, the router would automatically handle opening the connection to the remote router and doing the name translations.

As the design of the Active Templates planning concepts and tools evolved, a decision was made to go with a central master server instead of straight peer-to-peer communications, and so the remote protocol and code were mostly discarded (though it still exists, unused, in the delivered code, and the client-related code was never cleaned up as there was little demand for the Java client outside of ISX at the time). AXMP 1.1 as a protocol was discarded and the new 2.0 protocol used to build the new MetaRouter capabilities from the 1.0 baseline.

The MetaRouter was developed as an instance of the existing Router, using the UniqueNameMessageServer capability to prevent multiple instances of connections from the same machine. It was expected that all clients would connect to a single router on their local machine, which would then talk to the MetaRouter and the central TIM. The client side implementation was also updated to support the new *to* and *from* syntax, although these changes were far less drastic than the changes done for the 1.1 protocol.

The XSL transformation layer was added to this, and implemented as a subclass using a utility helper class, which allowed this feature to be an option at runtime, since it did add a lot of weight to the distribution.

The need for synchronous communication, matching request with reply, came from work being done by SoftPro, Inc., who was then experimenting with WebServices as a means of handling the communication needs. Their implementation used a simple SOAP protocol for passing messages, which in their server would then forward those messages as if it was an AXMP client. This created a problem because the SOAP protocol is inherently asynchronous; messages could be sent from the ContextManager or other components that had little to do with the request from the client. To solve this, the AXMP 3.0 protocol was developed. Clients and Servers now had the ability to add an "ID" tag to the request, and the messages in reply to those requests would include the same value in a "replyID" field, and could then include their own ID tag so that a conversation could take place each in reply to the other.

No official protocol changes were made beyond the 3.0 specification. The ability to rejoin an accidentally lost connection to the Router should the network fail, or to change the meta-router to use HTTP or some other protocol to address network issues was often

discussed, but there was never enough demand for this to warrant spending any time on protocols or implementations.

The final improvements in the Router were developed to improve the speed and robustness of the tool. Test cases were developed using Junit to hit the router for high stress and test that all messages being passed were actually getting through to the client. Out of these tests and some observations from researchers at SoftPro, Inc., it was discovered that when large files were being passed around, the communication channels between the Router and other peers was halted. Internally at the time, the Router had only one "worker thread" for sending outgoing messages, in order to make sure that no client received two interweaving messages at the same time, violating protocols. This was deemed to be too inefficient. To fix this, the Router added the MessageForwardThread as an implementation of a WorkerThread design pattern. When this was done, a unique thread was created for each client and the central Server in order to synchronize communications with that recipient but leave the rest of the Router open to handle more requests. If large files were sent, they would slow down the channels among the recipient, but other components could continue to talk to each other without delay.

3 Tool Interchange Manager (TIM)

The Active Templates Tool Interchange Manager (TIM) grew out of the Plan Manager developed early on in the Active Templates program. That tool managed a plan in the form of a Plan and Goal Graph. Applications could connect with it via a socket, and pass messages back and forth to inquire about, create, modify, and manipulate nodes in a Plan and Goal Graph, and their attributes.

The first iteration of the TIM was released on 17 July, 2000. It expanded and improved upon the Plan Manager. While the Plan Manager's messages were text and resembled XML, they were not truly XML. The 17 July release sent and received messages in an XML format. The PlanManager was only able to manipulate one plan at a time. The 17 July release was able to store and manipulate multiple plans, each under an umbrella structure called a "Context". It responded to requests for lists of contexts, creating a context, switching contexts, etc.

Separate from this effort, a group of planning tools, called SOFTtools, had come into being. These applications implemented collaboration via a file sharing mechanism. This proved problematic in cases where sharing a disk wasn't possible. The 17 July release offered the ability to save and retrieve files – essentially a file sharing service, without the necessity of a shared disk. This facility was implemented to help ease the transition of existing SOFTtools applications to the larger Active Templates program. The saved files were organized by Context, so, once again, the Context was used as an organizational metaphor.

By October of 2000, the Active Templates program had decided not to pursue further the Plan and Goal Graph concept. The next release of the TIM, on 16 February 2001, reflected this. The Plan and Goal Graph facilities were removed. The Contexts and Files

were retained, though, and more functionality was added. The Router, which is described in another part of this document, made its first appearance in the 16 February 2001 release. Also added was the capability to add Monitors. Monitors are user-developed callbacks which are invoked when specified messages are received by the Plan Manager. In this way, application developers can add their own server-side functionality to the Plan Manager without requiring recompilation or even re-linking.

The next release, on 11 May 2001, connected to a database via ODBC. Through the TIM, applications could pass SQL queries through to the database, and the results were returned in an XML format. Thus, plans could be made persistent via a database mechanism. Since all applications passed through the TIM, there were no database contention issues, and applications could use the simple TIM messaging protocol rather than adapt to ODBC. Changes to database access could be absorbed by the TIM without requiring changes to applications.

With the next release, 27 July 2001, User and Role data structures were added to the TIM. Also, several new components were added. Most of these were existing ISX tools which were simply adapted to the TIM's messaging scheme. The Query Mediator provided schema-independent access to databases, and the Web Scraper provided query-like capabilities into web pages. A Workgroup manager was added, which simply monitored users and workgroups.

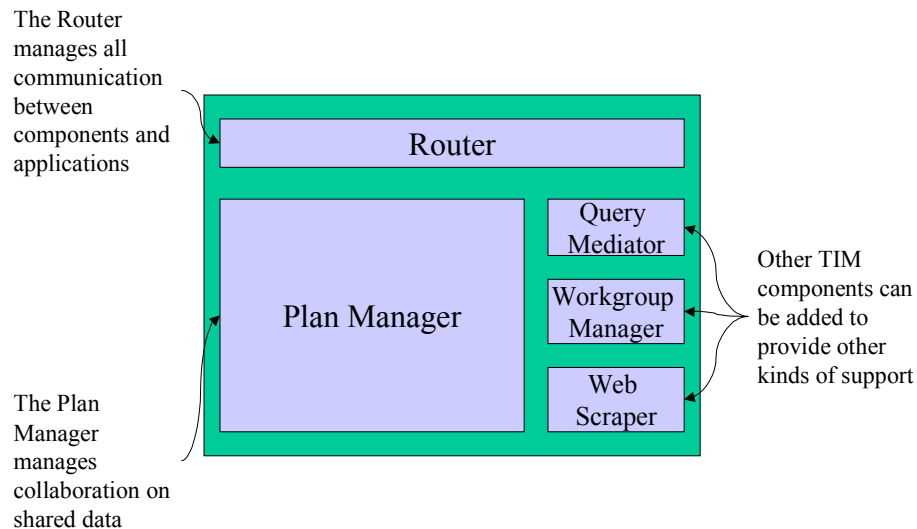


Figure 1 – Tool Interchange Manager (TIM)

A decentralized architecture has advantages over a centralized one. A client-side component was therefore envisioned, and development begun. Applications would connect to their local infrastructure component (called the "Application Layer" or AL),

which would then manage the connection to the server. This simplifies the job of the application developers, since it places the burden of server operations (locating, connecting, uploading from, downloading to, merging with, etc.) on an infrastructure piece. The infrastructure piece can also manage login, so that each application doesn't force the user to re-assert his/her identity.

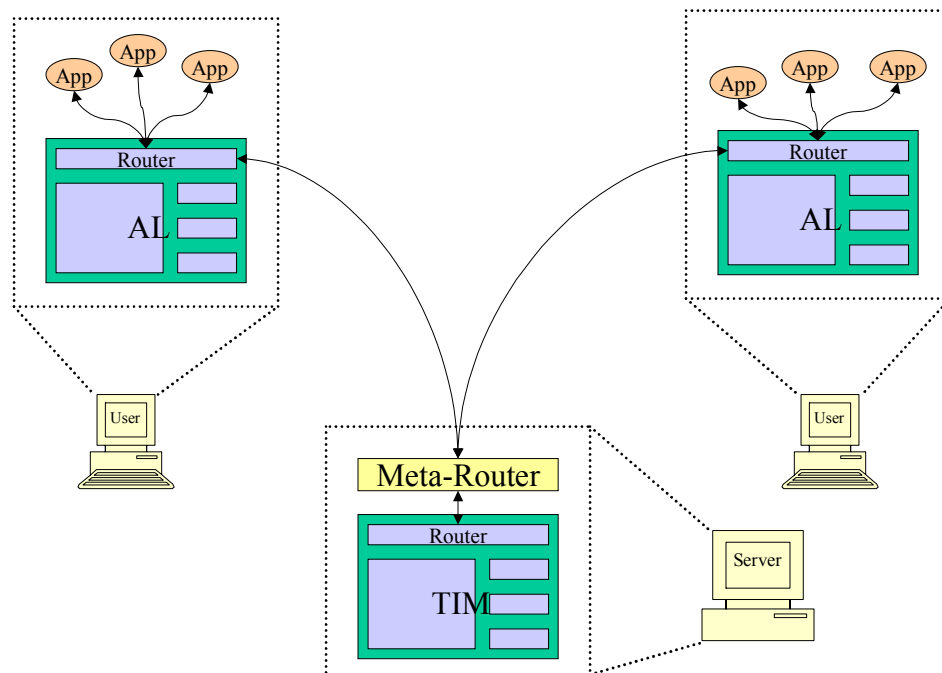


Figure 2 – Application Layer (AL)

The AL could act as its own mini-server, so that applications could interoperate even when separated from the server, or could even form a small enclave, with one AL taking the role of the TIM. The AL and the TIM would, therefore, have very similar structures, and so they were built from the same code base. The 4 September 2001 release was the first in which the user was given the option of installing in AL mode or TIM mode.

In order to manage communications between the TIM and the ALs, the Router was made more sophisticated, and a MetaRouter component was introduced. Each machine – both server and clients – had a Router to manage all of the components on that machine, and the MetaRouter handled communications between Routers, to get messages between machines. A more sophisticated addressing scheme was necessary, and was introduced.

Also appearing with the 4 September 2001 release was a detailed User's Guide. This 60-page document discussed each of the components of the TIM and the AL, and their capabilities and uses. It also included references for all of the XML messages application developers would need.

The next two releases, on 31 October 2001 and 20 November 2001, were incremental improvements on the previous version. Each fixed bugs and added small pieces of functionality to the TIM/AL suite of tools. One major piece of functionality was added to the Router with the 31 October 2001 release – Stylesheets. Application developers could install XML Stylesheets on the Router to transform messages into an XML format which might be easier for their applications to parse.

The 8 March 2002 release added specific messages and a schema to support a joint ISX and SoftPro experiment. This schema described Terminal Aerodrome Forecasts (TAFs) used by SoftPro’s WeatherWrite and other SoftPro tools.

The TIM/AL concept was eventually deemed to be too complicated, but there was still need for a client-side infrastructure presence, so a user application called the Work Center was developed. The Work Center allowed the user to view and manipulate all of the objects maintained by the TIM, to directly query the database, to log in at a single point, to chat with other users, to see which applications (and which versions of those applications) were attached to their TIM, and other such monitoring functions. The Work Center was first available for the 22 April 2002 release of the TIM, and then for the final 31 May 2002 release. There was an interim release on 17 May 2002, but this was an alpha released intended only to allow the developers at SoftPro to work with new constructs that were being hardened for the 31 May 2002 release. The Query Mediator, Web Scraper and Workgroup Manager were removed.

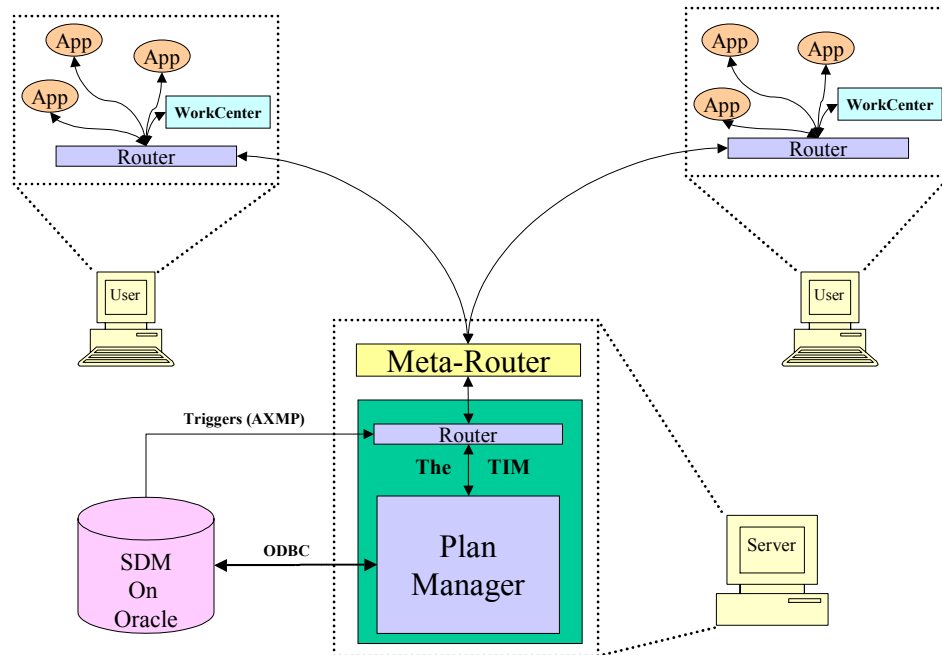


Figure 3 – Work Center

These releases also introduced more sophisticated installation procedures, which were easier for users. There were, in fact, two installers, one for the server-side suite of tools,

and one for the client-side Work Center. A great deal of documentation was also developed for this release – the original User’s Guide was renamed the Developer’s Guide to better match the role it served, and a true User’s Guide was developed, describing the use of the Work Center. A System Administrator’s Guide was also developed, which explained how a System Administrator could install and uninstall the server, change ports used, and otherwise configure the system.

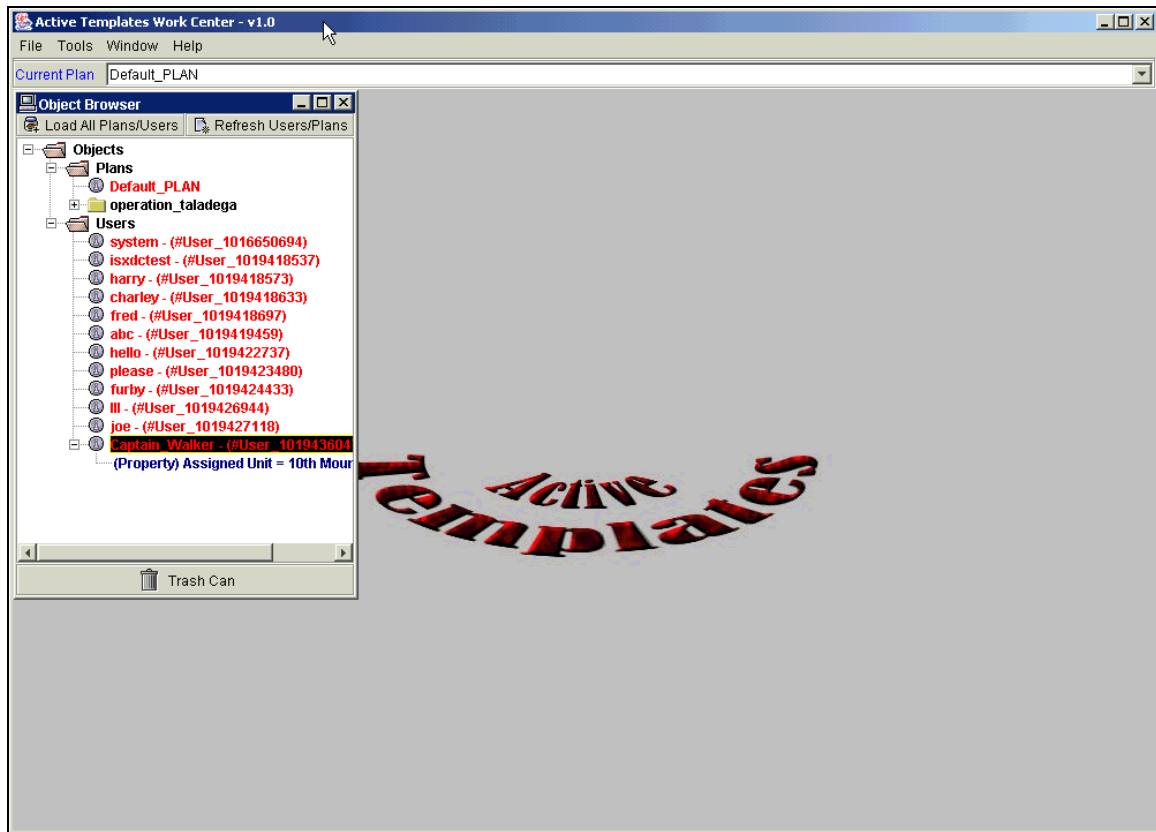


Figure 4 – Work Center Window

Also incorporated into the 22 April 2002 release was the Structured Data Model (SDM). The TIM used SDM creation scripts to instantiate an SDM in Oracle. It also installed triggers in the Oracle database, so that any changes to the tables of the SDM were sensed, and event messages were sent by the TIM to all applications working in the same context.

After the June 2002 Active Template program meetings, the entire TIM concept was determined to be too complex for the project, and the TIM was dropped from the project after the 31 May 2001 release. Thus, the 31 May 2002 release is the final release of the TIM.

4 Structured Data Model

The Structured Data Model (SDM) was designed to provide a shared representation and repository for planning data. A relational database was used to provide the repository

while a relational schema was designed to represent the data. This model was to provide simpler interoperability for components of the Active Templates (AcT) program compared to pair-wise integration between applications. In the end, integration was not as straightforward as planned. This document describes the evolution of the SDM and how it was integrated by the AcT community.

The first version of the SDM, designed by Larry Lafferty of SoftPro Technologies and Mark Peot of Rockwell Scientific was released in Spring 2002. The initial version was implemented using Microsoft SQL Server and later ported to Oracle 8.1.7, the target database protocol for the customer.

Their goals for designing the system were as follows:

- Accommodate all of the requirements for shared planning data from other AcT application developers.
- Minimize the number and complexity of the tables.
- Support distributed transactions.
- Support record-level security markings.
- Space efficiency.
- Computational efficiency.

Many of their goals for the SDM were achieved by virtue of the power of the relational database system. Other goals were supported in the design of the table structure. One of the key aspects of their design was not to duplicate data that was expected to be deployed in the customer database. Therefore, valuable information, such as personnel data, was not included in the original design. User readability was not a focus of the design as database naming constructs were inconsistent and views were not included in the design. Though this seemed reasonable during design, these problems were often brought up by users of the model.

The SDM originally consisted of approximately 45+ tables that can roughly be categorized in 6 groups: documents, plan elements, time tables, map objects, constraints, and environmental data. Documents were top-level entities referring to files, web pages, or actual documents, such as legacy planning files. Plan elements are any logical portion of the plan, such as mission, subplan, fire support plan, etc. Each of these elements had a start and optional end time. Plan elements could have child-parent or peer-peer relationships. AcT times represented projected and actual times used in the model. These times could be either relative or absolute, represented as seconds since Jan 1, 1970. Map objects described any object that could be referenced on a map. Constraints consisted of some rudimentary temporal checks.

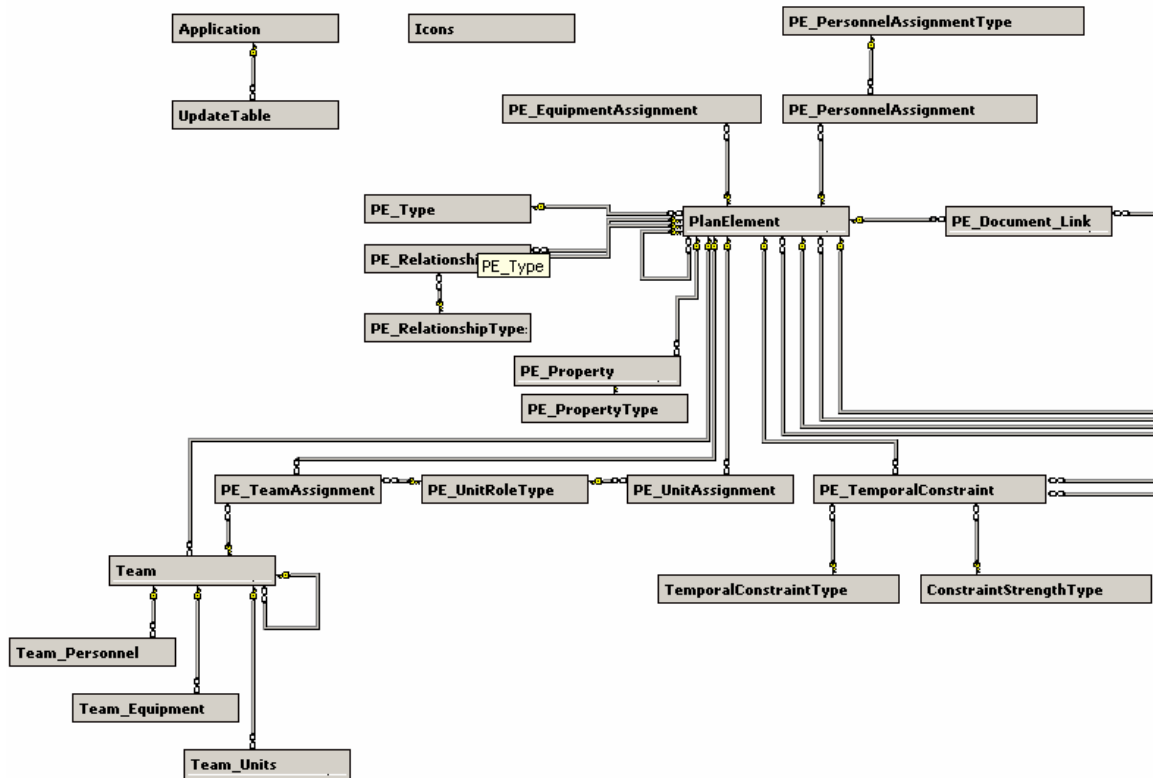


Figure 5 – Initial SDM Plan Elements and Related Tables

Some special design idioms were used in the original SDM to improve data integrity and accountability of data. Globally Unique Identifiers (GUIDs) were applied from the OSF Distributed Computing Environment to serve as keys for all objects entered by multiple users. Though software is available for generating GUIDs, none became widely accepted by SDM users. The Tool Interchange Manager (TIM) provided this service, but other infrastructure pieces that followed it did not. GUID creation became a point of contention with developers integrating into the model. Update identifiers were also introduced to track which users insert records into the database. This idea of tracking through a modular fashion has merit, but many developers did not like the tracking scheme and may have preferred a database-centric solution.

The SDM was built to be flexible for all data representations, such as XML. In supporting the idea of property tables, where attribute value pairs could populate the table, the DII-COE XML registry syntax was used to define domain values. In this way, any datatype could be defined according to an accepted standard. This particular standard turned out to be difficult to use and was eventually ignored by developers.

By the time of the Summer of 2002 AcT PI meeting, ISX Corporation had inherited responsibility for maintenance of the SDM. They integrated their TIM infrastructure components to use the latest model. Meanwhile, General Dynamics Advanced Information Systems (GDAIS) had created a theoretical mapping from their .SOF file format from SOFTools to the SDM.

After the Summer PI Meeting, ISX had the singular task of integrating applications into the SDM, including supporting the needs of applications that could not yet represent their data into the model. The focus of future enhancements would center upon representing and managing constraints, with the help of Jim Blythe at USC/ISI, and resources, with support from Alice Mulvehill of BBN and Steve Smith from CMU. Incremental improvements to the SDM would continue with additions such as point of contact (POC) information.

ISX began integrating AcT researcher's tools with the original design of the system. The C2PC Tools group at Global InfoTek was one of the first groups to integrate their data into the SDM. Their application desired the use of monitors of the data they would produce. GITI loaded their data through the TIM mechanism and the concept of operations was to have other groups monitor their data. The integration of this data led to many fixes to the original model. Though the amount of data was very small, it only took a few days for their tool to interact with the TIM and SDM.

Based on the opinion of many of the AcT developers, the design of the SDM did not support the necessary constraint ability that is critical for many AcT applications. For sake of simplicity, the SDM leveraged Jim Blythe's constraint model as used in his tool, CONSTABLE. Adding constraints to the SDM was not as straightforward as one would have hoped as the complexity of the representation did not fit well into the relational model. The term constraint has many meanings when applied to a data model. Here are four varieties of constraints that were used in the SDM:

- Database Constraints – The relational database has a built in ability to do type checking, primary keys, referential integrity, etc.
- Database Property Constraints – The database model allows for property/value pairs and a “data type definition” is given using the XML DII-COE Registry syntax. These constraints are not explicitly checked by the database.
- Dynamic Constraints – These are any constraints that aren't defined in 1 or 2. These are the types of constraints that could be defined by arbitrary data. This can be done by users or done with some initial database of constraints.
- Constraint Advertisements – This is the “capability” of a constraint.

The SDM's design included a table structure for these explicit constraints. The following is a simple description of how constraints could be used from the SDM:

- A “constraint-checking application” advertises some constraint capabilities.
- Someone enters actual constraints for fields in the SDM based on constraint capabilities that are advertised.
- A user enters data into a field that has a constraint.

- A “constraint-using application” sees the constraints, and looks up the advertisement.
- The advertisement is acted on by the “constraint-using application” and the results are formed by “constraint-checking application”.

The constraint section of the SDM was composed of four tables. The first was the advertisement which basically stores an XML constraint capability and an application. Two other tables are representing actual constraints; this is a representation of the XML from “Constraints in Active Templates” (<http://www.isi.edu/expect/projects/temple/constraint-definition.pdf>). Constraints have any number of arguments, and the capability exists to constrain over literals, database columns or actual instances. Components that use constraints can only define constraints that can be solved by some program. This limitation may seem troubling, but it enables the application of constraints without human intervention. The fourth table is Constraint_Advertisement_Links. This would associate Constraints with their appropriate advertisements. Every constraint would have available software to “check” the constraint.

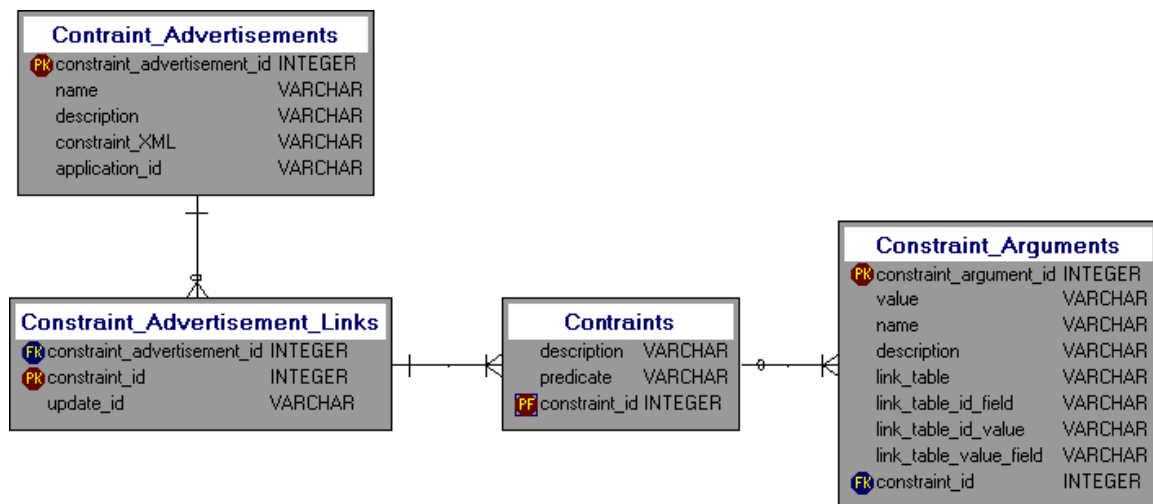


Figure 6 - Constraint Tables w/o links to rest of SDM

The resources portion of the SDM represented instances of resources, types of resources, and allocation of resources, possibly to plan elements. Much of the data found in these tables was likely to be available in the unit’s database. These tables were created to facilitate integration and would eventually become views upon the unit’s database.

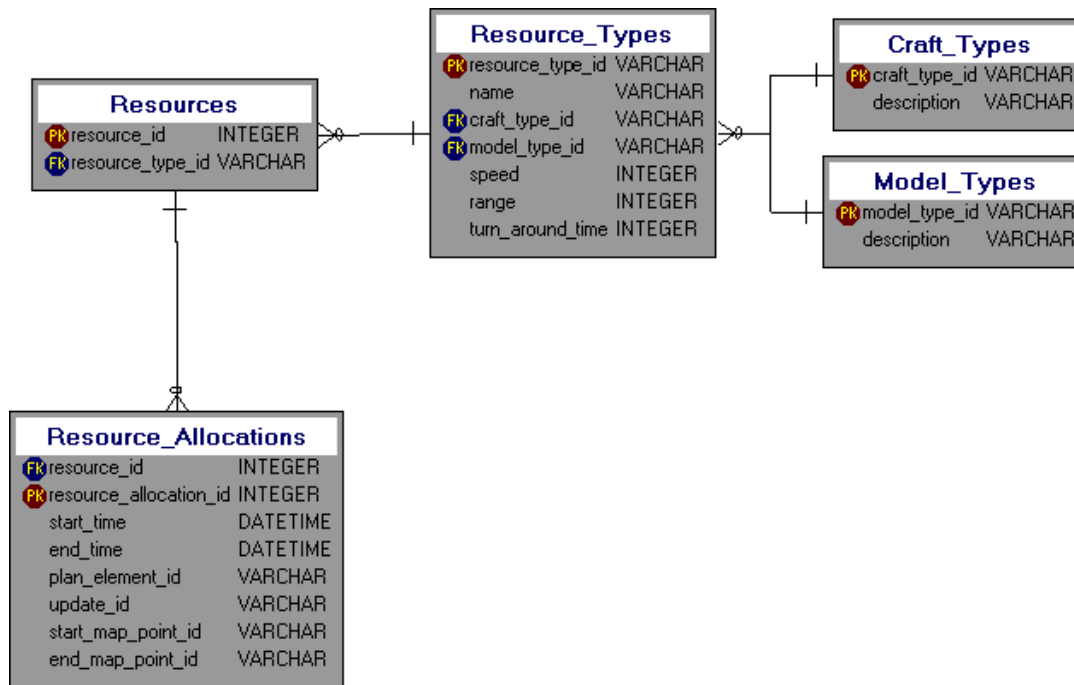


Figure 7 - Resource Tables (without links to rest of SDM)

GDAIS introduced the DBProxy in September of 2002. The goal of the DBProxy was to provide an interface and database for hosting the SDM. DBProxy replaced the functionality of the TIM-line of infrastructure components. This application was a HTTP interface to a database, defaulting to an SQLite database. This tool also contained a version of the SDM that was not entirely consistent with the SDM supported by ISX.

The introduction of the DBProxy tool led to confusion among those supporting enhancements or currently integrated to the ISX maintained SDM. At this point, there were two versions of the SDM available. There were also a myriad of ways to access the databases and an array of different databases supported.

Due to the lack of consensus on the SDM issue by members of the AcT community, it was determined that a radically different approach was needed. The divergence in schemas and complexity of the systems had shown the need for a single SDM that was easy to understand. The idea was to create an initial schema that was extremely simple - ultimately 9 tables with no referential constraints - and incrementally build upon that design as necessary. The resulting design supported all of the content of a .SOF file, but was much simpler for a human to digest and understand than previous SDM approaches. This design also leveraged MySQL as its relational engine. MySQL is free and widely used.

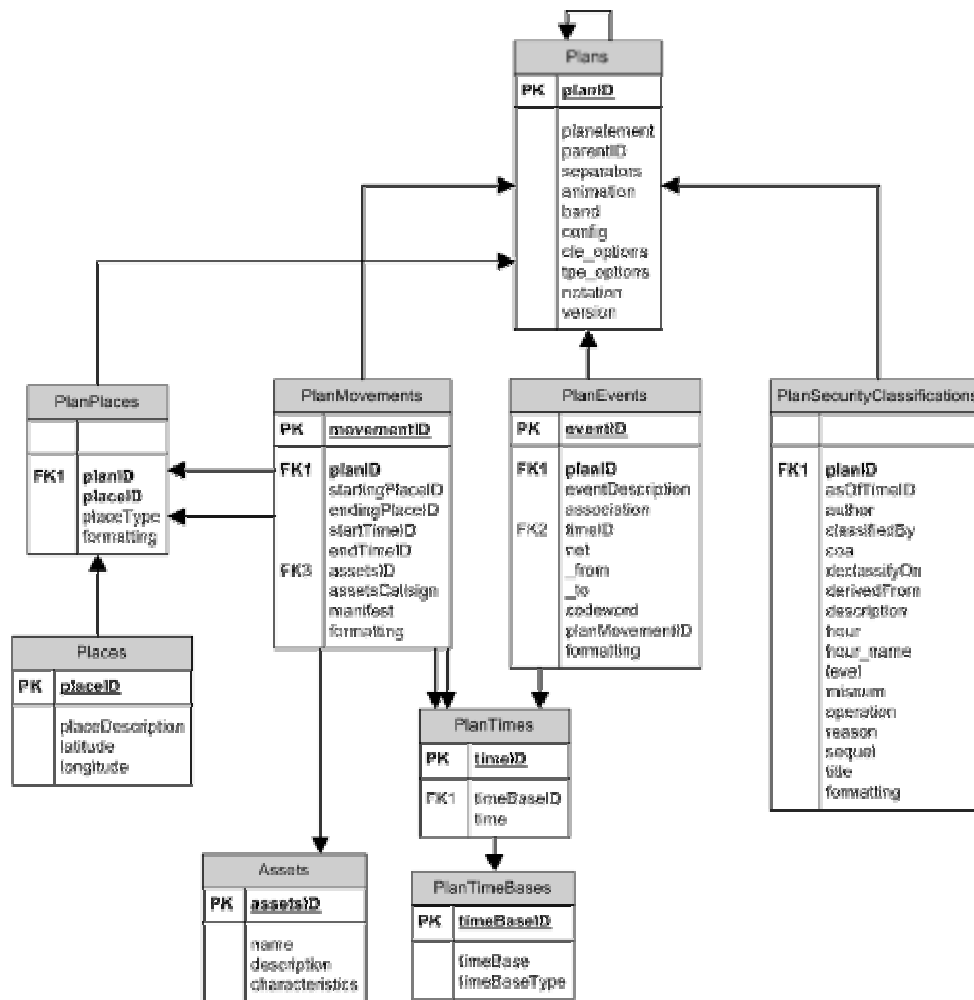


Figure 8 - Simple SDM with implicit foreign relations

By the time of the November 2002 AcT PI meeting, Rockwell and GDAIS were using the DBProxy SDM and BBN and ISX were using the “Classic” SDM making it clear that complexity was only part of the issue for SDM integrators. It did not seem obvious that the tradeoffs for the Simple SDM made it a viable option for future integration.

A number of alternatives were considered and explored during the development of the SDM. This small list provides some insight to other technologies that were used or considered.

- The .SOF file is the format used by the SOFTools planning tool. The format of this file was geared for quickly deriving the visualization for SOFTools plans. This format was widely adopted by the community as a de facto-standard. This format has some limitations; geared toward representing many graphical details rather than factual details. Also there are very obvious limitations with dealing with file systems including concurrent writing and file discovery. The .SOF format always provided a fallback for integrators if they chose not to use the SDM.

- Web services have increasingly become a powerful way to share the power of applications across a community. Much of the standards in this area have only been developed in the past few years. No web services have been explicitly been deployed by the AcT community, however many AcT applications do provide services that could fall under this umbrella.
- One of the limitations of the relational database was the complexity in creating rich structures. Of particular problem was the hierarchy of plan elements. XML was perceived as solving many of the representation issues and the next logical step was XML Databases. While these databases may have solved some of the problems, the standards are not completely set and limited options exist for supporting tools.

The use and development of the SDM was key to its acceptance or lack thereof. A few reasons have been identified as perceived candidates:

- The SDM was too complicated for its intended use.
- The learning curve of the SDM was too high for developers to make the commitment.
- The integration of tools into the SDM was too difficult compared to alternative means of integration.
- The SDM seemed to be continuously changing or have divergent versions.
- There was a lack of widespread adoption in the community.
- Developers were not ready to integrate.

All of these led to an SDM that was not widely used, many of these follow common difficulties in designing large scale database schemas among users with different models of the data.

5 Plan Coordinator

After the TIM efforts were halted, ISX turned its attention to developing a Plan Coordinator tool. This tool, suggested by Fred Bobbit, provides a higher-level view of a plan. It allows a higher-level commander to view each component of his plan, and assign a point of contact to each. He can also view the SOFTTools plans generated for each component of his plan. (Note: this tool is NOT a parallel representation of the contents of a .SOF file, it is instead a tool to organize multiple .SOF files.) Each component can have annotations, where different personnel can record their requirements for others. The Plan Library allows a commander to compose a plan with previously composed pieces, rather than component-by-component. The associated Gantt chart shows the user a timeline view of his plan.

This tool was begun after the June 2002 meetings, and was completed and delivered in December 2002.

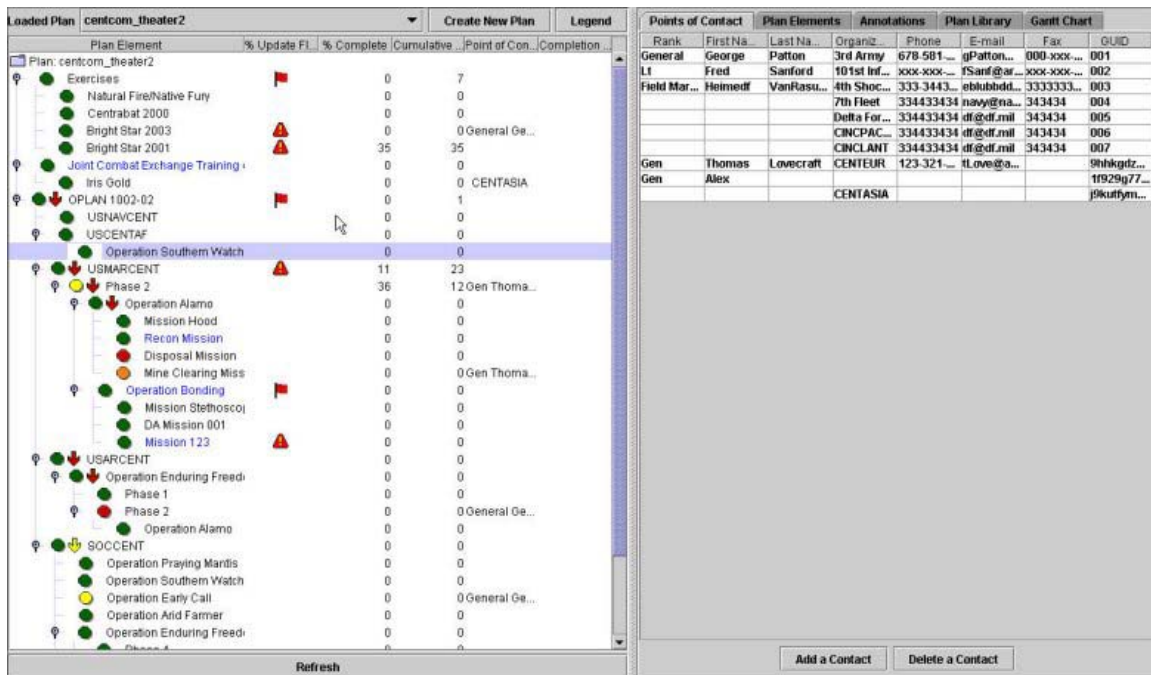


Figure 9 – Plan Coordinator

6 Lessons Learned

ISX built predominantly infrastructure for Active Templates. Unfortunately, there was no mandate for application developers to use that infrastructure. The application developers often built to their own specifications, and viewed it as a risk to change. From the application developers we got many divergent requirements, such that it was impossible for ISX to satisfy them all.

This use of a capitalist, market-driven model for technology development was not successful. Indeed, it is our belief that this model cannot succeed within a DARPA R&D program. The reasons for this are many but one critical issue is the reluctance of researchers to base their applications, which they are trying to position for reuse with future programs or commercial application, on software that they themselves do not control or have full knowledge and rights to.

ISX had much difficulty in establishing and maintaining links to the research and operational communities. These communities eventually sought, and found, standardization guidance, i.e., the SDM, from elsewhere in the Active Templates program.. ISX also did not do a good enough job establishing or maintaining relationships with the R&D development teams and managing their expectations.

Appendix A – Jumpstart

A-1 Introduction

ISX, SoftPro and ASI worked together in the early stages of the Active Templates program to create a planning toolkit, and collaborated with GITI to demonstrate this toolkit. The toolkit is capable of representing, storing and maintaining plans. It allows multiple applications to access and modify the plan, and keeps all applications informed of any changes. It is also capable of maintaining relationships between the attributes of plan elements. The plan structure and relationships between attributes are captured in flat files, and not a part of the software structure. Thus, this knowledge may freely be changed without the need to recompile or relink the software. Capabilities from other software elements may be included, and even controlled, from the toolkit by means of Pluggable Planner, described later. In addition, a simple spreadsheet-like interface allows users to directly view and manipulate the plan.

This document primarily describes the products of the work performed by ISX. The work of the other companies involved has been described here to the extent necessary to explain the ISX work. For more details of the contributions of SoftPro ASI and GITI, please refer to their reports.

A-2 Plan Representation

Plan and Goal Graphs (PGGs) were chosen as the underlying plan representations for plans managed by the planning toolkit. ISX, ASI and SoftPro have all had many years of experience with this technology, and the concepts of Skeletal Planning which grow out of it. The nodes with attributes lend themselves to the concept of Templates, and the hierarchical nature of PGGs lends itself to the concept of nested templates. One of the fundamental strengths of Skeletal Planning is its ability to reason with incomplete information and to react to incoming changes, all of which are well suited to Active Templates.

A Plan and Goal Graph is a layered structure, with two kinds of nodes: Plans and Goals. Goal nodes represent states of the world which are desired to be achieved. Plan nodes represent actions that will achieve the goals. The layers alternate: all children of a Plan node will be Goal nodes, and all children of a Goal node will be Plan nodes. The Goal children of a Plan node represent subgoals, ALL of which must be achieved for the plan to be feasible. It is an "AND" relationship. The Plan children of a Goal node represent plans to achieve the goal, ANY ONE of which must be sufficient to achieve the goal. It is an "OR" relationship. Thus, a PGG is an AND-OR graph. Both Plan nodes and Goal nodes have Attributes.

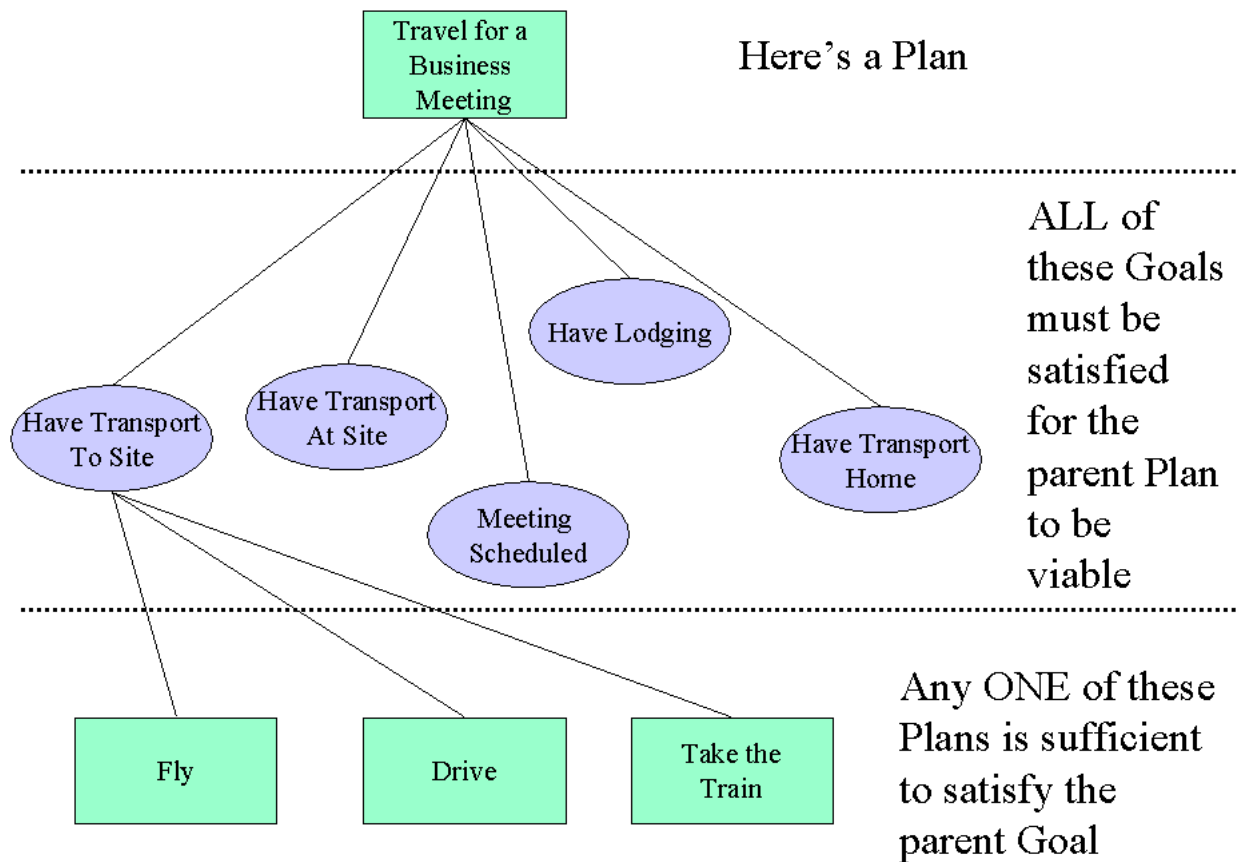


Figure A-1 Plan Goal Graph

The structure of a plan is captured a priori in a Pattern graph. Nodes in a Pattern graph represent structural elements of a plan, but are general and do not represent any particular plan. They are prototypes, and analogous to Classes in Object-Oriented design. The Pattern graph is captured before planning begins through knowledge engineering. In the process of planning, an Instance graph is created by instantiating nodes of the Pattern graph and giving values to their attributes. The Instance graph is the representation of a specific plan, and is the primary structure manipulated by the Plan Manager.

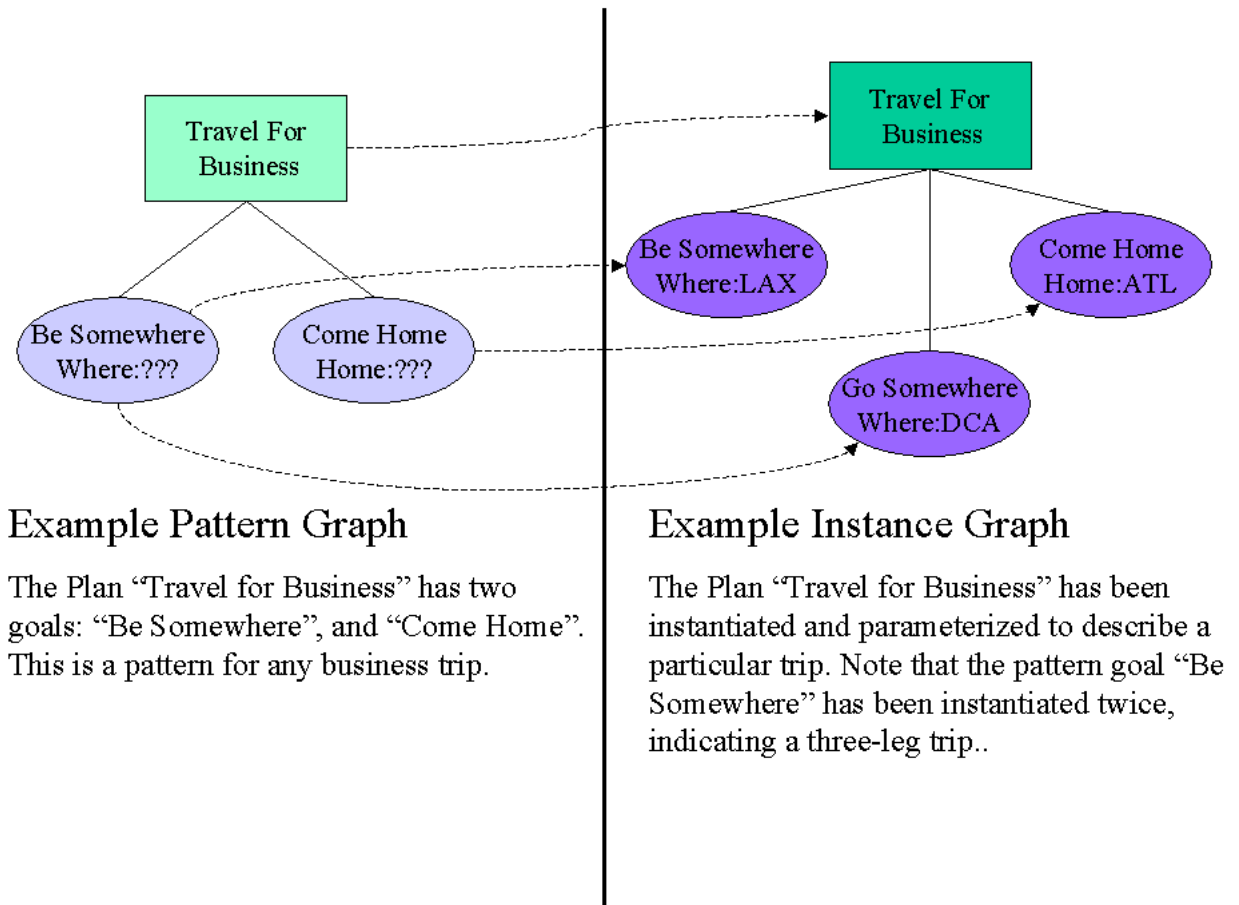


Figure A-2 Instantiated Plan Goal Graph

A-3 Architecture

The toolkit system developed under the Jumpstart work consists of several interacting components. The components are designed to all reside on the same machine. The Plan Manager is the central component, being responsible for the maintenance of the plan. It is built on top of ASI's PGG libraries. The GUI Coordinator allows multiple applications to use the Plan Manager via a socket-based connection. All applications are able to manipulate the PGG, and the Plan Manager informs all its customers of any change made by any application. The Spreadsheet GUI is a general user interface which is closely tied to the GUI Coordinator. GITI's map-based interface uses the GUI Coordinator to communicate with the Plan Manager. There are several Pluggable Planners and a Preference Reasoner, all of which are made available to the Plan Manager through Microsoft Dynamically Loading Libraries (DLLs).

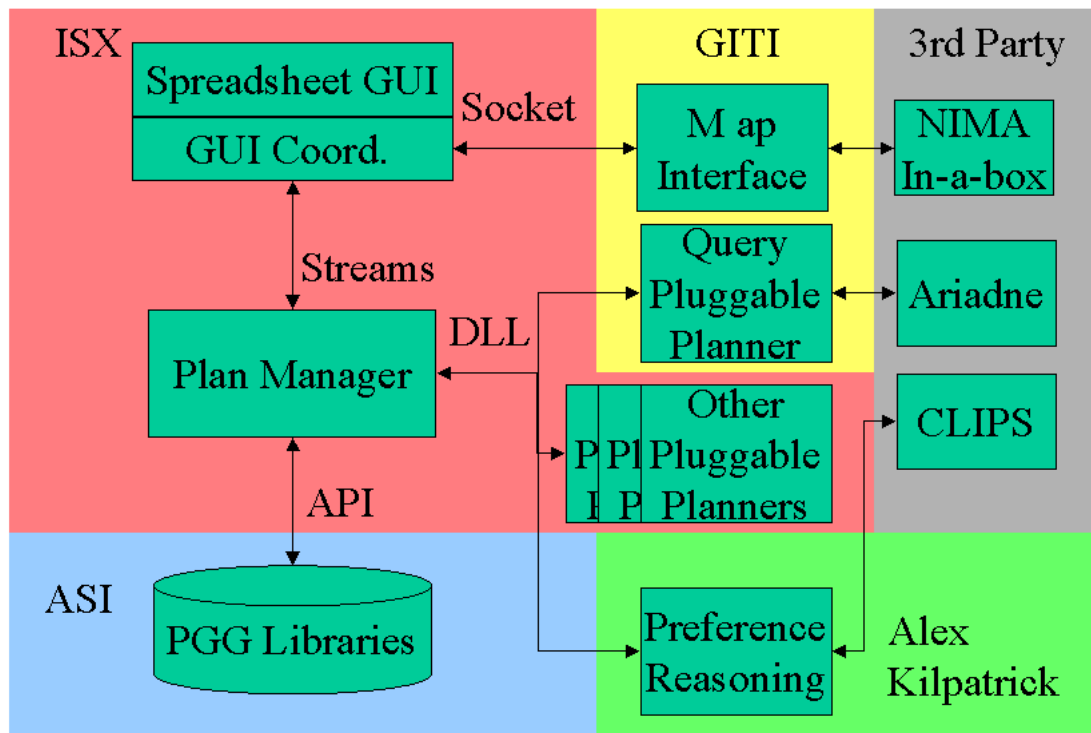


Figure A-3 Plan Manager Architecture

A-3.1 Plan Manager

The Plan Manager is the primary component of the toolkit. It manages the plan, in the form of a PGG. It allows applications to manipulate the plan, and informs all applications if any makes a change. The Plan Manager is written in MS Visual C++ 6.0.

In addition to the nodes of the PGG, the Plan Manager manages relationships between the attributes. There are three kinds of such relationships between attributes:

- Preferences are suggestions made by the Plan Manager based on business rules specified in CLIPS. They are not considered as actual values until the user explicitly accepts them.
- Constraints are relationships between attributes which are either true or false. The Plan Manager does NOT enforce that constraints be satisfied – it informs the user of any constraints which are violated, but allows planning to continue. Constraints are "Pluggable", meaning that developers can install outside software to compute constraints and configure the Plan Manager to use that software without recompiling or relinking.

- Dependencies have three parts: a set of input attributes, a set of output attributes, and a function to perform. There are two kinds: Automatic and Manual. Automatic dependencies are invoked by the Plan Manager automatically whenever conditions are right (all the attributes exist, all inputs have values, no new attribute values will be overwritten, etc.). Manual dependencies are made available whenever conditions are right, but must be explicitly invoked by the user.

A-3.1.1 Underlying Representation

The Plan Manager is built on top of the ASI libraries, and uses their PGG representation. The Pattern Graph is specified in a series of flat text files, which can be created and edited via an ASI tool, or may be modified by hand with a text editor.

An example Goal:

```
GOAL:
  PatternID: HaveLodging
  Recurrence: SINGLE_OCCURRENCE

  // class defining

  // instance defining
  Attribute: goal_name CHARSTR HaveLodging

  // updateable
  Updateable: MaxLodgingCost VAR *MaxLodgingCost*
  Updateable: TotalLodgingCostAtDestination VAR
    *TotalLodgingCostAtDestination*
END_GOAL
```

An example Plan:

```
PLAN:
  PatternID: UseCommercialLodging

  // class defining

  // instance defining
  Attribute: plan_name CHARSTR UseCommercialLodging

  // updateable
  Updateable: CheckInMonth VAR *CheckInMonth*
  Updateable: CheckInDate VAR *CheckInDate*
  Updateable: CheckOutMonth VAR *CheckOutMonth*
  Updateable: CheckOutDate VAR *CheckOutDate*
  Updateable: LengthOfStay VAR *LengthOfStay*
  Updateable: Smoking_NonSmoking VAR *Smoking_NonSmoking*
  Updateable: NumberOfOccupants VAR *NumberOfOccupants*
```

```

Updateable: KingStandard VAR *KingStandard*
Updateable: HotelName VAR *HotelName*
Updateable: DistanceFromAirport VAR *DistanceFromAirport*
Updateable: HotelPhone VAR *HotelPhone*
Updateable: HotelRewardsNumber VAR *HotelRewardsNumber*
Updateable: Rate VAR *Rate*
Updateable: LodgingCost VAR *LodgingCost*
END_PLAN

```

The link between them:

```

LINK:
  PatternID: Link_Pattern_10
  Parent: GOAL HaveLodging
  Child: PLAN UseCommercialLodging
END_LINK

```

In addition to this structure, the relationships between attributes are specified, again in flat text, in another file. Here is an example:

```

Auto basic numberofdays
Plan UseCommercialLodging
In MONTH1 CheckInMonth
In DATE1 CheckInDate
In MONTH2 CheckOutMonth
In DATE2 CheckOutDate
Out DAYS LengthOfStay

```

This is an Automatic dependency, in the file “basic.dll”, performing the function “numberofdays”. It is attached to the plan “UseCommercialLodging”. It has four inputs: MONTH1, DATE1, MONTH2 and DATE2, which are mapped to UseCommercialLodging’s attributes CheckInMonth, CheckInDate, CheckOutMonth and CheckOutDate. There is one output, DAYS, which is mapped to UseCommercialLodging’s LengthOfStay attribute. The same function can be used many times in many places. The following is an example of the same function as above being used in a different part of the PGG:

```

Auto basic numberofdays
Plan DriveRentalCar
In MONTH1 PickUpMonth
In DATE1 PickUpDate
In MONTH2 DropOffMonth
In DATE2 DropOffDate
Out DAYS NumberOfDays

```

In these simple examples, all of the attributes were located in the same node. There is a path specification mechanism (which will be described later under “Application Interface”) by which attributes from multiple nodes may be included in a dependency.

There are also mechanisms for specifying that an attribute may not be necessary, or that there may be zero or more repetitions of a node (and, thus, its attributes).

All of the domain knowledge is contained in these flat files. The software is completely domain-independent, and can be easily (and radically) reconfigured by editing these files.

A-3.1.2 Attribute Maintenance

Along with maintaining the structure of the Pattern and Instance graphs, the Plan Manager also affects relationships between attributes. Any change by a user is sent to the Plan Manager in the form of messages, described below in the section "Application Interface". With any change, the Plan Manager implements an impulse of processing which invokes dependencies when necessary. The Plan Manager first resolves dependencies, then constraints, then preferences. Of these, only dependencies change actual values, so it is safe to check preferences and constraints afterward without fear of needing to fire more dependencies.

A dependency is viable if all of its input attributes and output attributes exist (i.e. their nodes have been instantiated) A viable dependency is firable if it can be used as an Automatic dependency. There are some stringent rules. All of its input attributes must have been given values at some time; otherwise, it is passing along non-information, with possibly bad results. None of its output attributes can have been given a value in this impulse - otherwise, there may be infinite loops. These two rules ensure that no damage is done, but it must also be assured that there's a "good reason" to fire this dependency - that some value will be gained, that some new info will be passed along. Thus, SOME input must have changed this impulse (which means that there's some new information to be passed along), OR there must be SOME output which has never been given a value.

In more detail: An attribute is considered specified if the user has given it a value, or if it has gotten a value through dependencies operating on other specified values. Note that preference values are not considered specified until the user explicitly accepts them. A specified attribute is marked if it has been given a value in this impulse. An Attribute is fresh if it has never been given a value. If any input attribute is fresh, then firing this dependency would be propagating undetermined information, so this dependency is NOT firable. If any output attribute is marked, then it's already been given a value in this impulse. Giving it another value isn't useful, and indeed may lead to an infinite loop, so this dependency is NOT firable. If neither of the above is true, then when is it useful to fire a dependency? If any output attribute is fresh, then giving it a value is useful, so the dependency is firable. If any input attribute is marked, then it's gotten a new value in this impulse, and passing that new info along is useful, so the dependency is firable.

Putting this all together, a dependency is firable if:

- No input attribute is fresh
- No output attribute is marked

- Some input attribute is marked OR some output attribute is fresh

Whenever a change is dictated by the Spreadsheet Interface or some other application, any attributes which have changed values are marked and their new values passed along to the Preference Reasoner. This begins an impulse. Then, the Plan Manager collects all viable automatic dependencies. It then determines which are firable, and, one at a time, fires them (i.e. computes the output attribute values from the input attribute values via its function). This causes the output attributes to be changed, and thus marked and passed along to the Preference Reasoner. This, in turn, may cause some other viable dependencies to become firable (and some other to become NOT firable.). The new firable dependencies are fired, and so on, until no other viable dependencies are firable. After resolving all dependencies, constraints are checked in the same way. Constraints cannot change attribute values (they only have input attributes), so there is no danger of additional dependencies becoming firable. Then, the Preference Reasoner is checked to see if any new preferences became available. If so, these values are given to the attributes, but only if the attributes are not yet specified. Preference values are themselves not considered specified, so again, there is no danger of additional dependencies becoming firable. At the end of the impulse, all attributes are unmarked.

A dependency is usable if it can be used as a Manual dependency. The rules are a less stringent. There's no concept of an impulse, since a manual dependency is fired by the user, not automatically by the system. There's no need to check if there's a "good reason" to fire this dependency - that's the user's decision. The Plan Manager must simply make sure that every input has been given a value (i.e. that no input is fresh.)

A-3.1.3 Pluggable Planners

The toolkit uses drop-in DLLs to include outside functionality. Such a DLL is termed a "Pluggable Planner." To include outside functionality, a developer need only follow these steps:

- Implement one of the following functions, and export it from a DLL:
 - For dependencies,


```
void performFunction(const char *function,
                    const NamedValue *inParams, NamedValue
                    *&outParams);
```
 - For constraints,


```
int checkConstraint(const char *function,
                   const NamedValue *inParams);
```

In either case, NamedValue is defined as follows:

```
typedef struct { CString name; CString value; }
NamedValue;
```

- Modify the dependencies file to specify which attributes are affected by the new functionality. The new section tells where to attach the dependency (i.e. which node), what DLL it's in, which function within the DLL, and input and output

parameters. There is an example above in the section “Underlying Representation”.

- Put the .dll file in the same directory as manager.exe, which is the Plan Manager executable.

A-3.1.4 Preference Reasoner

The Preference Reasoner was developed as a C++ DLL by Alex Kilpatrick. It uses CLIPS, a public-domain inference engine, to specify and implement business rules. The Preference Reasoner is told about new attribute values on every impulse, and responds with any new computed preferences. The following is an example rule:

```
(defrule DC-Hotel-Preference "In DC, we want Key Bridge"
  old-fact <- (node $?root-path1
    UseCommercialLodging?ucl-id HotelName "NULL")
  (node $?root-path2 TravelToDestination ?ttd-id
    DestinationLocation "Washington, D.C.")
  =>
  (assert (node $?root-path1 UseCommercialLodging
?ucl-id
    HotelName "NULL"))
  (assert (preference $?root-path1
UseCommercialLodging
    ?ucl-id HotelName "Key Bridge Marriott"))
)
```

A-3.1.5 Application Interface

Applications communicate with the Plan Manager by opening a socket stream with the GUI Coordinator, and then sending and receiving text string "events". These ASCII strings specify the actions the application can take, and also the results of those actions.

At startup, the PM will instantiate a root instance node, and any of its children it deems necessary. It will inform the GUI of these newly instantiated nodes. Henceforth, as nodes are added or attribute values changed, the PM will send messages to the GUI. The GUI can send messages to the PM directing changes to the graph. Each message is on a single 'line', terminated by a carriage return. This goes both ways - the messages coming out of the PM will be so formatted, and messages into the PM should be so formatted.

The PM will send the following messages to applications:

<Plan ...>	upon instantiation/change of a Plan node
<Goal ...>	upon instantiation/change of a Goal node
<Attribute ...>	when an attribute changes

Before describing these in detail, it will be useful to define the following sub-component:

```

<AttributeValue
  <Value "value in quotes">
  <Specified [Yes|No]>
  <ConstraintViolation [Yes|No]>
>

```

NOTE: This would be all on a single line - they have been split here for clarity. This describes a value of an attribute. The value is always a string, and is always wrapped in quotes. The Specified field indicates whether the user has specified a value for this field. The value will either be the string "Yes" or the string "No", without quotes. Likewise, the ConstraintViolation field indicates whether this attribute value is involved in a constraint violation. Now, the messages:

```

<Attribute
  <Name [attribute name]>
  <Node [path to node]>
  <Necessary [Yes|No]>
  <Fixed [Yes|No]>
  <CurrentValue <AttributeValue ...> >
  <PossibleValues
    <AttributeValue ...>
    <AttributeValue ...>
    ...
  >
  <Planners
    <Planner [planner name]>
    <Planner [planner name]>
    ...
  >
>

```

This message plays two roles. This is the message sent when an attribute changes, and it is also sent as part of the <Plan ...> and <Goal ...> messages. Different parts may be sent or not sent, depending on context. Name is the name of this attribute, and is always needed. Node is the path to the node to which this attribute is attached. It's needed if this is a standalone message describing an attribute change, but it's not needed if this Attribute message is embedded in a message describing the Plan or Goal this Attribute is a part of. Necessary is a boolean, indicating whether or not this Attribute must have a value in order for its node to be considered complete. It should rarely change. Fixed is a similar boolean, indicating whether this attribute's value is fixed (i.e. not changeable by the user) CurrentValue should be self explanatory, as should PossibleValues. The GUI is expected to maintain its own list, so this is only for startup. It may be eliminated in future releases. Planners is a list of the names of planners which may be used to compute the value of this attribute.

A new or changed Plan or Goal is described using these messages:

```

<[Plan|Goal]
  <ID [instance ID]>
  <Prototype [prototype ID]>
  <Parent [path to parent]>
  <Attributes
    <Attribute ...>
    <Attribute ...>
    <Attribute ...>
    ....

```

```

>
<Potentials
  <Potential <ID [id]> <Title "title in quotes"> >
  <Potential <ID [id]> <Title "title in quotes"> >
  <Potential <ID [id]> <Title "title in quotes"> >
  ....
>
>

```

ID is the Instance ID of this Plan or Goal Instance node. Prototype is the ID of the Prototype (or Pattern) node which this node is an instance of. Parent is the Goal which contains this Plan, or the Plan which contains this Goal. Attributes is a list of this node's attributes, each in the format of the Attribute message described above. Potentials is a list of potential children. For Goals, it is a list of plans which may be used to satisfy this goal. For Plans, it's a list of Goals which may be added to this Plan. The IDs are strings which identify the potential child to the Plan Manager. They may be Instance IDs or Pattern IDs, but that is not important to the GUI. The GUI simply sends this string to the Plan Manager in an `<addGoal...>` or `<selectPlan...>` message, both of which are described later. The Title is a slot to put a pretty name, so the GUIs can put a prettier face on it. The value is a quote-delimited string.

Here are the messages which the PM can receive from the GUI:

```

<setValue <Attribute [path to attribute]>
  <Value "value in quotes"> >

<selectPlan <Goal [path to goal]>
  <Plan [plan prototype name]> >

<invokePlanner <Attribute [path to attribute]>
  <Planner [planner name]> >

<addGoal <Plan [path to plan]> <Goal [goal prototype name]> >

<getGraph>

```

Looking at them individually:

```

<setValue <Attribute [path to attribute]>
  <Value "value in quotes"> >

```

This message sets a value of an attribute. The value is always a string, and is always in quotes.

```

<selectPlan <Goal [path to goal]>
  <Plan [plan prototype name]> >

```

This message selects a plan to satisfy a goal. Note that the plan is identified by its pattern name. This should be one of the names in the `<PotentialPlans ...>` list for this goal.

```

<invokePlanner <Attribute [path to attribute]>
  <Planner [planner name]> >

```

This message invokes a planner, which will compute a value for this attribute (and perhaps some others). The planner name should be taken from the <Planners ...> list in the Attribute message.

```
<addGoal <Plan [path to plan]> <Goal [goal prototype id]> >
```

This message adds a Goal to a Plan. The [goal prototype id] must come from the <Potentials...> list of the Plan node.

```
<getGraph>
```

This message requests a copy of the entire Instance graph.

Paths are complete descriptions of the location of a node or an attribute in the Instance PGG, starting from the root. The path alone should be sufficient to find the node or attribute from the root of the PGG.

The paths that the GUIs will be dealing with are in the Instance graph, and so they are paths through Instance nodes, composed of Instance IDs. Instance IDs are built in the following manner: start with the Pattern ID, put a Pound sign ('#') at the front, and append an underscore and a number at the end. The first instance of a 'Fly' pattern node would be called #Fly_0, the second would be #Fly_1, and so on.

The path to a node is a complete path from the root to that node, separated by colons (':'). The root is just a colon. Plans and goals alternate. So, the plan to Fly under ArriveAtDestination might look like this

```
:#TravelForBusiness_0:#ConductBusinessAtDestination_0:
  #TravelToDestination_0:#ArriveAtDestination_0:#Fly_0
```

Where : is the root goal, #TravelForBusiness_0 is the top-level plan, #ConductBusinessAtDestination_0 is a Goal, with #TravelToDestination_0 as its plan, etc.

The path to an attribute, is the path to the node it's in, followed by a dot ('.'), and then the name of the attribute. So, the path to the AirFare attribute in the Fly plan would be

```
:#TravelForBusiness_0:#ConductBusinessAtDestination_0:
  #TravelToDestination_0:#ArriveAtDestination_0:
    #Fly_0.AirFare
```

Here are some example messages:

```
<setValue <Attribute :#TravelForBusiness_0:
  #ConductBusinessAtDestination_0:#TravelToDestination_0:
    #ArriveAtDestination_0:#Fly_0.AirFare> <Value "208"> >

<selectPlan <Goal :#TravelForBusiness_0:
  #ConductBusinessAtDestination_0:#TravelToDestination_0:
    #ArriveAtDestination_0> <Plan Fly> >
```

Note that the ID of the Plan in this last message is a Pattern ID, not an Instance ID.

A-3.2 GUI Coordinator

The GUI Coordinator is a socket interface which applications use to communicate with the Plan Manager. The Coordinator receives messages in from applications and passes them in to the Plan Manager. It takes responses from the Plan Manager and routes them to all connected applications. The GUI Coordinator is developed in JAVA.

A-3.3 Spreadsheet Interface

The Spreadsheet Interface is a simple GUI which presents a straightforward representation of a PGG. It is based on Doug Dyer's initial concepts. Goals are considered the roots of templates, and the structure of the templates change as different plans are selected. With the Spreadsheet Interface, the user may view the PGG in a tabular form, and may make changes to attribute values. Color codes inform the user which attributes have been given values and which are only suggested, which attributes are involved in a constraint violation, which attributes are necessary for the plan to be considered complete and which attributes may not be changed.

The Spreadsheet Interface is implemented in JAVA, and is built atop the GUI Coordinator. Note that while it communicates with the GUI Coordinator directly rather than through a socket, it still speaks the same "language" as other applications, and thus could be easily split into its own applications. It is in the same JAVA virtual machine as the GUI Coordinator for the sake of efficiency.

A goal with a single plan is represented in the spreadsheet as follows. The name of the goal is treated like the name of an attribute, and the name of the plan is treated like an attribute value.

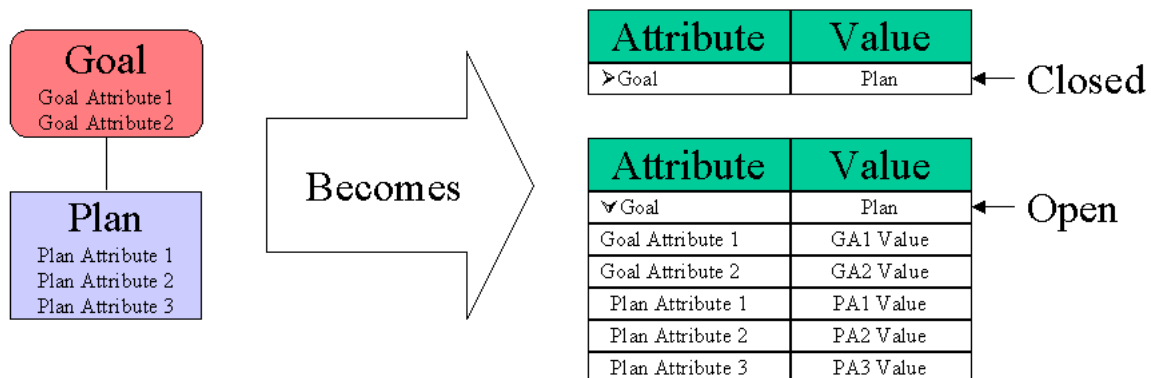


Figure A-4 Single Plan Representation

If a goal has multiple plans, then all of the plans are present in a pull-down menu in the value column. The structure of the template changes depending on which plan is currently chosen.

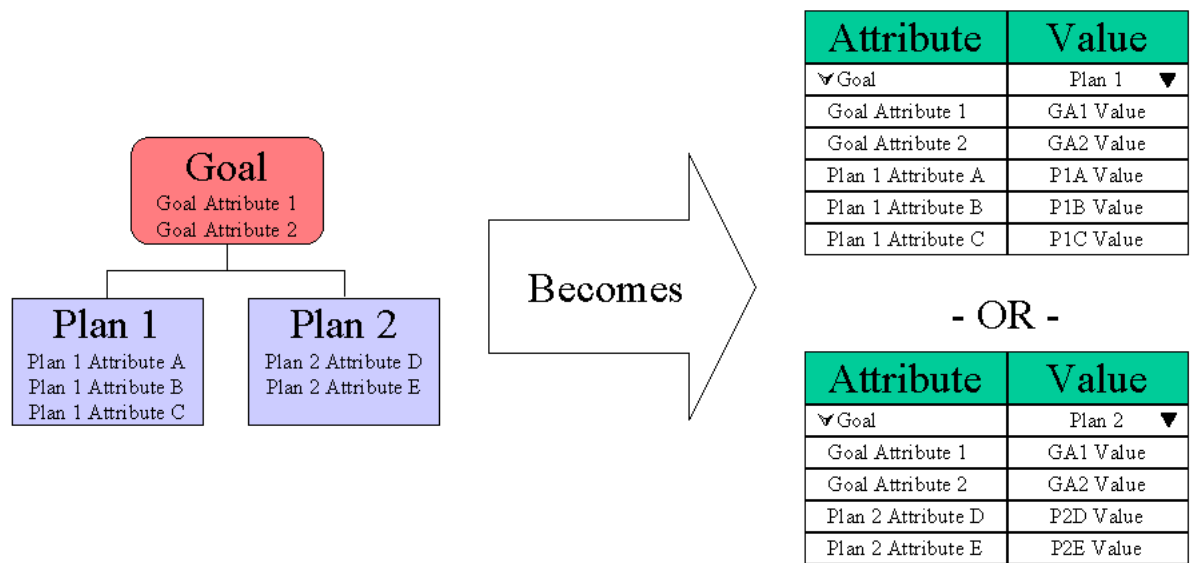


Figure A-5 Multiple Plan Representation

Goals which are children of plans are represented as nested templates.

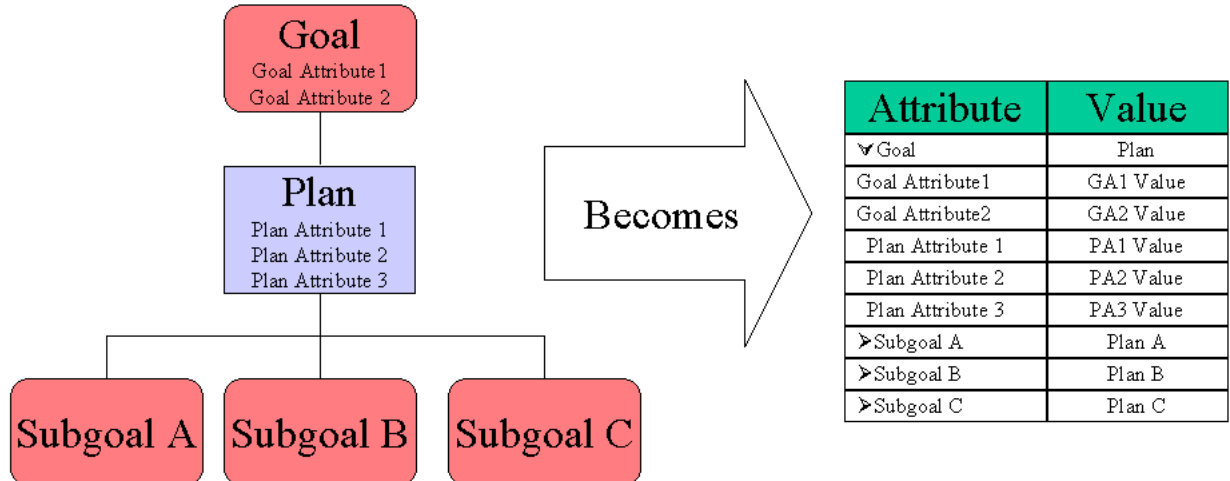


Figure A-6 Plan Children Representation

The following is a screen shot of the Spreadsheet Interface:

The screenshot displays the 'Airfield Seizure' application interface, which consists of three overlapping windows, each containing a spreadsheet-like table with 'Attribute' and 'Value' columns.

Window 1: Airfield Seizure

Attribute	Value
Root_Goal	Seize-and-Secure-Airhead
Airfield-ID	Montgomery-Airpark
Target-Latitude	12.6677
Target-Longitude	-35.4738
references	http://www.yahoo.com
Airfield-Secure---P3	Expand-Airhead
Egress---P4	Extract-TF-and-Precious
Ingress-P2	Assault-Airhead
Required-closure	H+45
H-hour	2330 Z
Staging-Base	Hurlburt AFB, FL
Staging-Latitude	30:25:20N
Staging-Longitude	86:37:06W
Distance	1012 km
Assault-CP-Established	Establish-Assault-CP
Each-Observation-Post-Established	Observe
Each-Observation-Post-Established	Observe
Label	???
Latitude	???
Longitude	???
NLT-Time	???
Assigned-unit	???
Estimated-start-time	???
Estimated-closure-time	???
Enemy-Fires-and-Elem-Neut	Engage-P2-TGTs-With-FS
Force-Inserted	Coord-Integrate-Sync-Arrivals
Key-Objs-Seized-and-Isolated	Clear-and-Isolate-Airfield-Elements
Log-Support-Estab	Sustain-Force
PreOps-Complete--P1	Deploy-Force

Window 2: Template: Assault-CP-Established

Attribute	Value
Assault-CP-Established	Establish-Assault-CP
Latitude	38:53:31N
Longitude	77:01:54W
NLT-Time	H+15
Label	Righteous Cause CP
Assigned-unit	*NONE*
Estimated-closure-time	H+10
Estimated-start-time	H+5

Window 3: Template: Each-Observation-Post-Established

Attribute	Value
Each-Observation-Post-Established	Observe
Label	Airpark-NE Corner
Latitude	39:59:59N
Longitude	78:04:48W
NLT-Time	H+35
Assigned-unit	*NONE*
Estimated-start-time	H+15
Estimated-closure-time	H+25

Figure A-7 Spreadsheet Interface

A-3.4 Pluggable Planners

Several specific pluggable planners were developed to support both the Travel Planner and the Airfield Seizure domain demos. Each implemented the function specified above in a drop-in DLL.

A-3.4.1 Basic Functions

The basic pluggable planner contains an eclectic mix of simple functions – sums, equality, multiplications, and the like. It is intended to capture a minimal set of "standard" functions, much like the standard functions of a spreadsheet. It supports the following functions:

- `equals` Set one attribute equal to another
- `multiply` Compute the product of a set of attributes
- `isum` Integer summation
- `dsum` Dollar summation
- `fsum` Floating point summation
- `numberofdays` Number of days between two dates
- `time+` Add an offset to a time, display in "H+[hours]" format. The offset is part of the function name, immediately following the plus with no embedded spaces.

- `startwith`: Give an attribute a default value. The default value is part of the function name, immediately following the colon with no embedded spaces.

A-3.4.2 Ariadne Queries

Ariadne is a web query engine developed by ISI. It essentially puts a SQL-like query language around web pages. ISI developed some queries for travel planning which accessed real travel-related web pages. GITI developed a Pluggable Planner interface so that these web queries could be used in the demos. The Ariadne Query Pluggable Planner supports the following functions:

- `airport` Find the airport code for a city
- `car` Find a rental car
- `hotel` Find a hotel
- `flight` Find a flight

A-3.4.3 Force Selection

The Force Selection Pluggable Planner was developed to support the Airfield Seizure demo. It contains the following functions:

- `assignment` Assign units to goals based on polygonal areas of responsibility
- `time+` Add an offset to a time, where the time is selected from a list via a key. The offset is part of the function name, immediately following the plus with no spaces.

A-3.4.4 Constraint Formulas

The Constraint Formulas Pluggable Planner provides a simple capability to specify constraints. Instead of a function name, the developer specifies a mathematical formula with a true or false outcome. A public-domain expression evaluator is used to evaluate the constraint. The following is an example of the specification of a simple constraint:

- Constraint formula `a<=b`
- Plan `DriveRentalCar`
- In `a` `PickUpDate`
- In `b` `DropOffDate`

A-3.5 GITI Map-based Interface

GITI developed a map-based interface for the Airfield Seizure demo. It was built in ArcView, using NIMA-in-a-box. The user was able to use map-based gestures to specify goals (e.g. road blocks, building seizures, etc.), landing zones, areas of responsibility, and other domain-specific information. This information was captured and relayed to the Plan Manager through the GUI Coordinator in the specified interface language. The user's actions became part of the plan, and the results were immediately viewable in the Spreadsheet Interface. Pluggable Planners were used for many actions, including the assignment of units to goals based on polygonal areas of responsibility.

Appendix B TIM Developers Reference Guide

B-1 Introduction

This document describes the features currently available in the Tool Interchange Manager (TIM) designed and built by ISX for the Active Templates project.

B-1.1 Features of the TIM

The TIM provides capabilities to allow applications to collaborate. It currently has two core components: the Router and the Plan Manager. There are three add-on components, the Workgroup Manager, the Web Scraper and the Query Mediator, which will be addressed in supplementary documents. The Router enables message traffic between applications, and between TIM components. The Plan Manager manages plan content and organization. Plan content is currently stored in files and databases, and is organized by contexts and by users.

There is also a client-side application called the Work Center, which provides a number of GUIs with which users and developers may visualize and manipulate the data in the Plan Manager. This tool requires user login, and may be used by applications to acquire the user's User-ID and Password without forcing the user to log in to each application.

B-1.2 Architecture

The TIM consists of two parts – a Router, which handles inter-process communication, and a Plan Manager, which handles shared data. These components reside on a server – however, there is also a router on each client machine, and a MetaRouter to manage message traffic between the clients and the server. Application developers need not worry about the location of the server – they connect directly to their local router, and allow it to manage the relationship with the server. The TIM manages access to the Standard Data Model (SDM), which is stored in Oracle.

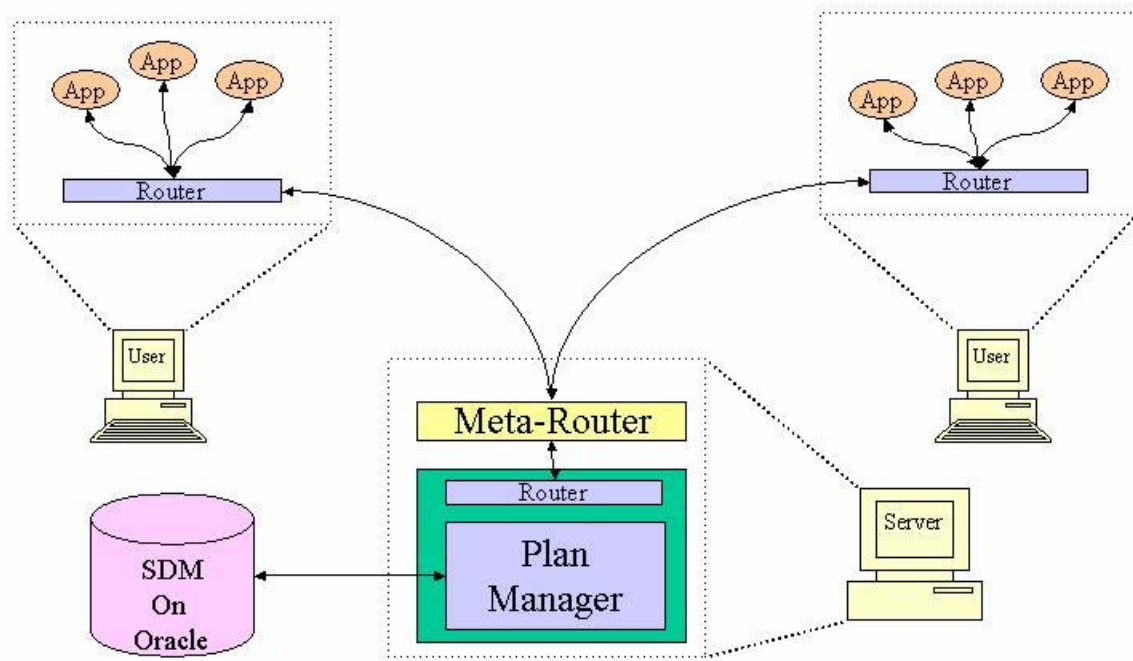


Figure B-1 TIM Architecture

B-2 Router

The TIM's Router manages the sending of messages between applications. It uses sockets, and has no client-side software to install, so applications need only open a socket (currently, socket 6226) and understand the Router's messaging protocol to communicate with other applications. The Router allows applications to send messages to each other, or to other components of the TIM. In particular, the Plan Manager component is handled slightly differently, as described below. The Router maintains a list of the connected peers, so that applications can always know what other applications are communicating on the same Router.

In the Active Templates topology, there will be a router on each client machine, as well as on the Server. These routers will communicate with each other through a MetaRouter on the Server machine. Applications need only connect to the Router on their local host – the client-side router will handle connections to the Server and other clients.

B-2.1 Messaging Protocol

The Router uses the **Active Templates XML-Based Messaging Protocol**, or AXMP. AXMP is a text based protocol, using an XML syntax, for passing XML messages between peers.

B-2.2 Connecting and Sessions

A client application connects to the router through a normal socket connection, on socket 6226 at localhost. For testing and experimenting purposes, one can use "telnet" to connect to the router. If using the built-in telnet for Windows, it is recommended that you turn "local echo" on in the preferences (and turn it off again later so you don't echo your password if you use telnet later on). Once connected, the client opens a session by sending the request

```
<request to="ROUTER" routerAction="beginSession" name="name"
class="class"/>
```

terminated by a newline, where *name* is the name of the application and *class* represents the type of application. By convention, the Active Templates project uses the vendor name/abbreviation in the class field. The server will respond by echoing "beginSession" as a message:

```
<message from="ROUTER" routerAction="beginSession"
to="class-name-####"/>
```

also terminated by a newline. The "to" field will tell the client its unique Router name. As shown in the example, this will be:

```
class-name-####
```

The #### is a four-digit number assigned by the router, starting with 1000 and increasing sequentially for each new connection. For example, ISX's Console application might send this request:

```
<request to="ROUTER" routerAction="beginSession" name="console"
class="isx"/>
```

It might be greeted with this response:

```
<message from="ROUTER" routerAction="beginSession"
to="isx-console-1001"/>
```

From this point forward, other applications on the local host may send messages to this application using the name isx-console-1001.

When a client is ready to terminate, it sends the endSession request:

```
<request to="ROUTER" routerAction="endSession"/>
```

and the server replies likewise with

```
<message from="ROUTER" routerAction="endSession" to="class-name-
####"/>
```

and closes the socket connection.

B-2.3 Requests Sent

Requests sent are wrapped in `<request>` tags, as follows:

```
<request to="address-list">
<!-- request goes here in XML, taking as many lines as necessary -->
</request>
```

Note that the `<request>` and `</request>` tags **MUST** each be on their own line!

The `address-list` is a list of one or more `address-specs`, separated by the vertical bar **OR** character (`|`). An `address-spec` is of the form:

```
name-spec@host-spec   or   name-spec
```

Where `name-spec` specifies the Router-assigned name(s) of application(s) to which to send this message, and `host-spec` specifies the host name(s) on which those applications reside. If the `host-spec` is omitted, the local host is assumed.

Both the `name-spec` and the `host-spec` may be explicit names of applications or hosts, or may be Regular Expressions, matching multiple names and/or hosts. Regular Expressions allow a single message to be sent to multiple recipients, possibly on multiple hosts. The Regular Expressions should follow normal POSIX regular expression rules, with the exception that the vertical bar **OR** (`|`) is reserved for separating entire `address-specs`. A short summary (from the regex engine being used) can be found at:

<http://jakarta.apache.org/regexp/apidocs/org/apache/regexp/RE.html>

Note the following about the behavior of the Router:

- The Router will NOT send a message back to the sender, even if the sender's Router name matches the `name-spec`.
- If the `@machine-spec` part of an `address-spec` is included, The Router will NOT send a message back to the local host, even if the local host's name matches the `machine-spec`.

to send to all "isx" apps	<code><request to="isx-.*"></code>
to send to all "arcview" apps	<code><request to=".*-arcview-.*"></code>
to send to all peer apps	<code><request to=".*"></code>
to send to a specific app	<code><request to="giti-arcview-1001"></code>
to send to an enumerated list of apps -- use the vertical bar <code> </code> as an OR: <code><request to="isx-spreadsheet-1001 giti-arcview-1002"></code>	

Table B-1 Request Sent

B-2.4 Messages Received

The Router receives requests, strips the `<request>` tags, and sends the content to the appropriate destination wrapped in `<message>` tags. Thus, ALL messages received are wrapped in `<message>` tags, as follows:

```
<message to="address-list" from="address-spec">
  <!-- message goes here in XML, taking as many lines as necessary -->
</message>
```

The `address-list` in the “to” attribute is precisely the `address-list` used to send the request, including any Regular Expressions. It is NOT merely the `address-spec` of the recipient. The `address-spec` in the “from” attribute is the `address-spec` of the sender.

B-2.5 Sending to a SERVER

The Plan Manager component of the TIM will be a special exception. Messages sent to `SERVER` on the local Router will get forwarded to the Plan Manager on a host called `TIM`. Responses, however, will come from `SERVER@TIM`.

The Plan Manager ONLY receives messages specifically intended for it – that is, the `name-spec` is precisely “`SERVER`”. It will NOT receive messages where the `name-spec` is anything other than “`SERVER`”, even if the `name-spec` is a Regular Expression which matches “`SERVER`”. The Plan Manager will receive messages addressed to `SERVER`, `SERVER@TIM`, or `SERVER@host-spec` where `host-spec` is a Regular Expression matching `TIM`, but this last case is dangerous, and should be avoided. The message will get sent to `SERVER` at every host matching the `host-spec`, and will get forwarded to `SERVER@TIM` from ALL of them, thus causing the real server to get multiple copies of the message.

Note the following examples.

Sending to the `SERVER`:

```
<request to="SERVER">
  <!--request to the SERVER goes here in XML form -->
</request>
```

Receiving from the `SERVER`:

```
<message to="address-list" from="SERVER@TIM">
  <!--The SERVER's message goes here -->
</message>
```

B-2.6 The Peer List and Peer Events

After the router sends the `beginSession` message, it will send the "peer list". This is a message that lists each of the clients already connected to that router. This list will include the client itself that is receiving the list, e.g.:

```
<message from="ROUTER" to="class-name-1000">
<peerlist><peer name="class-name-1000"/></peerlist>
</message>
```

The router will never send the full peer list again; it is the responsibility of the client to remember the list from that point on.

As new clients are connected or disconnected to the router, the client will receive a `peeraction` message informing it of the change, e.g.:

```
<message to=".*" from="ROUTER">
<peeraction peer="isx-querytool-1001" action="removed"/>
</message>
```

Again, it is the client's responsibility to keep track of this information.

B-2.7 How AXMP Differs from Normal XML

AXMP messages and requests are just like normal XML, with the exception of a whitespace requirement. The request tags sent by the client, both begin and end, **MUST** be terminated by newlines. The message tags from the router always will be so terminated. This is to help simplify the implementation -- the clients can examine the stream by a "readline", rather than having to examine every segment of the stream as it comes in for the presence of a tag. As an example,

```
<request to="SERVER@TIM">
<hi/>
</request>
```

is valid, but

```
<request to="SERVER@TIM "><hi/></request>
```

is not, and will get no response from the router at all. The `beginSession` and `endSession` requests/messages must be terminated by newlines as well.

B-3 Work Center

The Work Center is a client-side application, developed by ISX to help users and developers visualize and manipulate the data managed by the TIM. It has many useful debugging tools. Developers are encouraged to become familiar with the Work Center.

The Work Center will be a required part of a user's client, and users will be required to log into it before any work can be done. Applications will also be required to specify a User-ID and Password before accessing TIM capabilities – but, rather than ask their user, applications may send a message to the Work Center to retrieve the User-ID and Password the user has logged in with. It is still necessary for each application to login, but using the Work Center, they need not ask their user – the user can log in once at the Work Center, and applications can use this login info. For security reasons, the Work Center will NOT respond to requests from any Router other than the local Router.

B-4 Plan Manager

The Plan Manager manages plan content and organization. It allows applications to share content in a general way, rather than relying on developers to develop pointwise solutions between every pair of applications which wish to share data. The Plan Manager allows applications to read, add, and change plan content, in the form of data files and databases. It organizes content via contexts, and allows individual users to record user-centric information. It also provides an easy path for applications to add server-side functionality without requiring a recompilation of the TIM.

B-4.1 User Management

The Plan Manager supports the concepts of Users. Before accessing the SDM, each application must “log in” by supplying a User-ID and Password. These must match a legitimate User-ID and Password in the Oracle database. The Plan Manager uses the Oracle database for authentication of User-IDs and Passwords. Each Application MUST log in before accessing the databases. The User-ID and Password entered by the user are available through the console. Applications log in with the <login> message, as follows:

```
<login userid="userid" password="password"/>
```

Applications may obtain the User-ID and Password entered in the Work Center by the user by asking their local Work Center, as follows:

```
<request to="isx-workcenter-.*">  
  <getUserid/>  
</request>
```

Note that this request goes to the Work Center, NOT to the SERVER. The Work Center will respond:

```
<userid userid="userid" password="password"/>
```

Before using a User-ID, a User object must be created in the Plan Manager. Upon creation, the Plan Manager will check the Oracle database to see if the user already exists, and create it if it does not. The Console provides GUIs for these operations, and there are direct messages to the Plan Manager that applications may use to create Users.

B-4.2 Context Management

Contexts are essentially Plans. Each context holds a separate instantiation of the SDM (Standard Data Model). All of the objects describing a given plan are inside of a single context, and different plans are in different contexts. Contexts have names, and may be located by name. They also have Properties, and applications may use these properties to store and share information which is outside the realm of the SDM. Currently, the Plan Manager maintains a flat list of contexts, though in the future it may be desirable for contexts to be organized hierarchically.

There are several simple operations applications can perform with contexts. First, they can ask the Plan Manager for a list of contexts which have been created. They can then either select a context to work within, or create a new context. This is the equivalent of selecting which “plan” they wish to work with. Each application can operate in at most one context at a time. They will receive update messages for any objects in that context. Applications can also set properties on a context, so they can record any free-form data necessary. Before accessing any SDM information, an application must specify a context as follows:

```
<setContext contextName="contextName" />
```

While an application can operate in only one context at a time, it can receive change messages from other contexts by using the message subscription services. By subscribing to a context's message subscription list, the application will be notified of any changes that occur in that context. However, they will not be operating in that context and any queries that are sent only go to the context that you have set using the <setContext.> message. There is also a System Level message subscription service. This allows applications to receive notifications when another application successfully logs onto the system as well as when another application changes what context they are operating in. See the message definitions in the appendices for the syntax of the message subscription messages.

B-4.3 Access to the SDM

The Plan Manager allows full access to the Standard Data Model (SDM) through Oracle. Applications may make any legitimate SQL queries from these databases, including data changing actions like INSERT and UPDATE. Before accessing data, though, each application must first supply a User-ID and Password through the <logon> message. Then, the application must choose which plan to work in by selecting a context with the <setContext> message. After these two steps are complete, applications may then execute SQL commands on the SDM.

Applications wrap SQL commands in <query> tags, as follows:

```

<query queryID="unique-string">
  SQL statement
</query>

```

This sends a query through the Plan Manager to the database. QueryID is an id that the client assigns the query. The query results also have this field, to enable the clients to match the results with the query. There are several possible replies, depending on the type of SQL command executed, and whether or not it was successful.

B-4.4 Files

To simplify the transition of legacy software, the Plan Manager can act as a File Server. The File Server functions of the Plan Manager are centered around a context – that is, a context acts like a directory, containing the files. Applications can create text files, open them, read from them, write to them, and append to them. The Plan Manager will dispatch a message whenever a file is changed to all applications in its context, so there is no need for polling.

B-4.5 Files and Documents

Certain files have a specific role in the plan. In this case, they are called Documents. Existing files may be mapped to Documents, or new files added as Documents. The Documents are a part of the SDM, so a slot for a new kind of Document should be added through the SDM/SQL interface. In this way, Documents may be given metadata such as authorship information, classification information or modification dates, and may be associated with specific Plan Elements. Again, these operations are achieved by manipulating the SDM databases.

The Plan Manager uses its own File Server to store these files, and provides functionality to associate them with specific Documents. This is invisible to Applications. Applications need only send and receive content to associate it with a Document, using these messages:

```

<saveDocument peID="Plan Element ID" docID="Document ID">
  <![CDATA[File Content]]>
</saveDocument>

<getDocumentContent peID="Plan Element ID" docID="Document ID"/>

```

Plan Element IDs and Document IDs are available by querying the appropriate tables in the SDM, or via the <getDocument> messages provided. The above messages should be sufficient for most applications' uses of Documents. If, however, an application has already stored a file in a context, and merely wishes to assign that file to be a certain document without having to re-send its content, there are messages to support that, too:

```

<fileIsDocument file="filename" peID="Plan Element ID"
  docID="Document ID"/>

```

```
<getDocumentFile peID="Plan Element ID" docID="Document ID"/>
```

B-4.6 Monitors

Message Monitors are a way of adding functionality to the TIM manager without having to recompile the server. They use functions stored in a Dynamically Linked Library, called a DLL. This document will describe the different types of monitors and also explain how to use them.

B-4.6.1 The General Concept of Monitors

The power behind the concept of monitors rests in its flexibility and generality. A function or series of functions are stored in a DLL. A monitor is put into place and begins watching or “monitoring” messages. When a message being monitored is received, the monitor “triggers”, or runs the function in the DLL that it is using. DLL’s can be used to add functionality to the server without having to recompile the server. The monitor creation process is relatively simple, as will be demonstrated later in this appendix.

There are some restrictions placed on monitors. The function in the DLL only has access to a fixed set of the data that the server is using. It doesn’t have access to all of the data used by the server. Access to this data is controlled through something called a Parameter List. Pointers to the data are stored in these parameter lists, so when a function is created in a DLL, that function can only use certain pieces of data. This fact should not be cause for concern however, since most critical data is stored in the parameter lists.

B-4.6.2 Different Types of Monitors

The two types monitors are filtered and non-filtered. A filtered monitor can prevent message parsing from occurring. A filtered monitor’s function (in the DLL) must return a BOOLEAN value, either true or false, to state whether the parser in the server should go ahead and parse the message. If the function doesn’t return a BOOLEAN value, there is no way for the monitor to halt the processing of messages. If true is returned, the parser then parses the message and proceeds as though the monitor doesn’t exist. If the filter returns false, the message is not parsed, its simply discarded. A short example may prove useful for clarity.

A context in the server can store files and has file-processing messages to allow users to work with files. One of the things that a user can do is use the readFile message. This message returns the content of the file to the requesting user. Well, suppose a user wanted to add some simple security to a file that has been stored. This could be accomplished with a filtered monitor. Simply create a function that receives the sender’s ID and returns a BOOLEAN value in a user DLL. A user could write a function to check that particular ID against a list of ID’s that have access to that file. If that person’s ID is not in that list, the function returns false. If it is in the list, it returns true. Next, put the monitor in place and proceed. When the next user tries to read that file, the newly created function executes and checks that user’s ID. If the function returns false, the

readFile message is not parsed, and the contents of that file are not returned to the requesting user. If the function returns true, the message is parsed and the requesting user gets the contents of the file.

The other type of monitor is the non-filtered monitor. These monitors do not interrupt the parsing of messages. They are checked before the parsing starts. These are the most common monitors, due to the limited number of situations that require filtered monitors.

The two classifications, or levels of monitors are important as well. A description of these classifications follows.

A monitor can be attached in two ways. First the monitor can be attached to a context. Each context has its own parameter list, which is primarily comprised of data that is specific to a context. Examples of this would be file lists and property lists. A monitor may be attached to a context when a user wishes to monitor context-specific messages. Some messages in the DTD are context-specific. They do not actually have any meaning if not applied to a context. Some examples of context-specific messages are readFile, writeFile, getFileList, and createContext. Some messages are not context-specific, however, a context may still want to monitor them. The session message is a good example. It is not context-specific, yet a context may want to watch for this message and do some special processing when it receives it. A monitor attached to a context is called a context-specific monitor.

General monitors, or context unspecific monitors are attached to the context manager. The context manager holds and controls the contexts. When a user needs to monitor messages that are not context specific, like getContextList, this is the mechanism.

B-4.6.3 Message Monitor Composition

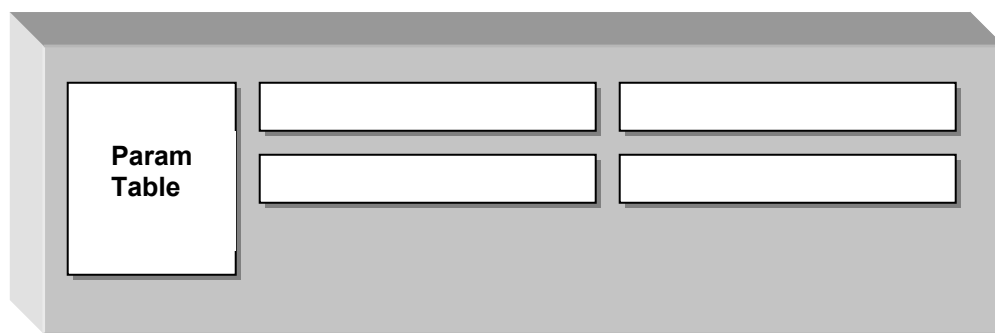


Figure B-2 Monitor

Figure B-2 illustrates the basic parts of the monitor. A monitor contains a table of values, called parameters, which are pointers to data in the server itself. This table holds the parameters needed by the function, not the complete list of available parameters (The complete list of available parameters is stored in the context or the context manager). The message to monitor is the message that the monitor is watching for. The function

pointer is the pointer to the function in the DLL that the monitor will fire when the message it's looking for comes along. The filter flag is the BOOLEAN value that tells whether or not this monitor is a filter. The active flag is another BOOLEAN value that is used to disable and enable monitors without having to remove them. These are the basic components of a monitor. There are messages that allow applications to change the filter flag and the active flag without having to add another monitor.

B-4.6.4 Creating a New Monitor

First, create the function in the DLL. The function looks like any other standard C function. The one difference to note is that all of the parameters that you use, which come from the parameter lists stored in the server, are pointers. This is true of all data types - even atomic types like integers are passed as pointers. The Monitor Function must dereference them in order to get their value.

```
Int * number;  
*number += 5;
```

In the monitor package you will find a MS VC++ workspace and project file that contains a template for creating DLL's. The DLL has a few key parts that must be present in order for the monitor to work. First is the DLL entry point. This is basically the "main" for a DLL. When the server calls the LoadLibrary function to load the DLL, this method is executed. Another important function is the setSender method. This method is called by the server when it creates a monitor. It sends the monitor a function pointer to the message-sending function in the server. This allows the monitor to use the server's message-sending method to send messages back to applications that are connected to the TIM. Both the DLL entry point and the setSender methods are already implemented in the DLL template provided. The last important piece is the EXPORT_FUNCTION macro. This is the mechanism by which the DLL tells the server what functions it has available for use. There may be functions internal to the DLL that don't get called by the server, but are merely utilities within the DLL. Such functions should not appear in the EXPORT_FUNCTION macro. EXPORT_FUNCTION should only be used on functions intended for the server to be able to call.

The EXPORT_FUNCTION macro defines the export syntax for the function. It adds the following:

```
extern "C" __declspec( dllexport )
```

This specifies that the function is exported explicitly. There are two types of exports, explicit and implicit. Implicit used the .lib that's created when you compile your DLL, and it uses fixed function prototypes in the calling application. Explicit exports mean that loading is done at run time, and no library file (.lib) is used. This allows the flexibility to be able to add functions at runtime without recompiling or re-linking the Plan manager.

Once the function is written, it must be declared for export in the header file. The function prototype must be added to the header file, with `EXPORT_FUNCTION` in front of it.

```
EXPORT_FUNCTION void myFunction( char * var1, int * var2 );
```

The only difference is the `EXPORT_FUNCTION` macro, otherwise it looks exactly like a normal C function prototype.

The next step is adding some information about the monitor to the `Monitor.cfg` file. This is the file that contains the information about any monitor that the server can use. There are a few fields in this file:

`PublicFunctionName` – This is the public name of the monitor. It's the name used in the `addMonitor` message.

`DllFunctionName` – This is the actual name of the function in the DLL. It is case sensitive.

`DllName` – This is the name of the DLL file that contains the function. The standard DLL that comes with the manager is called `Monitor.DLL`, but the system is not restricted to this one – developers may add DLLs as they deem necessary.

`Number of Parameters` – This is the number of the functions' parameters. The limit is 10.

`The Parameter List` - This is a list of the parameters, in order, that the function uses.

For example, assume a function called `myFunction` takes a `char *` and an `int *` as parameters. The function prototype may look like the following :

```
void myFunction( char * name, int * age );
```

The config file might look like this:

```
[MyFunction]
PublicFunctionName=MyFunction
DllFunctionName=myFunction
DllName=MyDLL.DLL
Number of Parameters=2
[MyFunction:Parameter]
Parameter_1=Sender
Parameter_2=Senders Age
```

Notice the parameter tags in the file. They don't say `name` or `age`. Instead, this is the key name for the desired data from the parameter list. This will be covered in the next section.

The final step is actually attaching the monitor. That's done via the addMonitor Message. For example:

```
<addMonitor monitorName="MyFunction" contextID="#Context_976821591"
message="getFileList"/>
```

If no contextID is given, the monitor is attached to the context manager and is considered a context unspecific monitor. This message puts the monitor in place. Whenever a getFileList message is sent to context #Context_976821591, this function will fire.

B-4.6.5 The Parameter List

The parameter list is a fixed set of information available to Message Monitors. The contents of the parameter list are different for the context manager than they are for an individual context. All individual contexts have the same parameter list. The parameter list is a hashtable, which stores a keyname for a parameter and a pointer to the location in memory where the data is being stored. When adding a new monitor to the Monitor.cfg file, the names in the parameter section are the key names for the data in the hash table. A complete list of parameter key names will be included further down. For example, consider MyFunction (defined earlier). It took two parameters, a char * and an int *. In the function itself (inside the DLL) the parameters were called name and age. In the config file, they were called Sender and Sender's Age. This is because in the config file, the values were the key names for the data in the parameter list. In the function itself, they were what the programmer wanted to call them. For example:

<u>Keyname</u>	<u>Pointer to data in memory</u>
Sender	0x00ff45
Sender's Age	0x0044ef

When the monitor is built in the server, it keeps a copy of the pointer to the data. When the monitor is triggered, it goes to that place in memory and retrieves the data located there and passes it as a parameter to the function.

When a monitor is created in the server, the server reads the Monitor.cfg file and retrieves that information. It needs that information in order to correctly create a monitor. It uses the parameter names in that file to get the data it needs to pass as function parameters. Below are the key names that are available as parameters for a monitor.

One final thing to note in the case of context specific monitors: the notion of properties. Contexts have properties, which are generic name-value pairs. A property name is defined by the user that creates it. The list of property names is available from a context via the getPropertyList message. In the context's parameter list, the key name is the property name.

Key Names	Data Type	Description
XMLMessage	Char *	The actual complete XML Message
HomeDirectory	Char *	The directory the server resides in. This is The base directory for context directories and the like.
ContextList	Vector *	An STL vector containing pointers to all of the Contexts the server has loaded.
Sender	Char *	The application ID of the originator of a message
NumberOfContexts	Int *	The number of contexts the server has loaded.

Table B-2 Context Manager Parameter List (context unspecific monitors)

Key Names	Data Type	Description
ContextID	Char *	The id of the context.
[Property name]	Char *	The value of a property
[Property name]	Char *	The value of a property
[Property name]	Char *	The value of a property
(repeat for each property)		
File	Char *	The name of a file
XMLMessage	Char *	The actual complete XMLMessage

Table B-3 Context Parameter List (context specific monitors)

B-4.6.6 Restrictions

There are a couple of restrictions. First, there is a maximum number of parameters allowed to the function in the DLL is 10. The function only has access to the data stored in the parameter lists. All parameters are passed as pointers, at this time there is no way to pass copies of data to a function. This will be corrected in a later version.

B-4.7 Utility Services

The Plan Manager also provides some other services. Many tables in the SDM use what is known as a Globally Unique Identifier or GUID. This is a string that is guaranteed to be unique and is used for the primary key fields of many tables. To insure that the GUID's that are created are in fact unique, the Plan Manager provides a GUID creation service for applications. You can send an XML Message to the Plan Manager and tell it how many GUID's to create and it will return a list of them to you for use in your application. It is suggested that you ask for 20-50 at a time and store them in your application so that you do not have to ask for a GUID everytime you do an operation in the SDM that requires a GUID.

Appendix C - Router

The Router built by ISX for the Active Templates program is a Message Oriented Middleware (MOM) server implementation, based around a protocol referred to as AXMP (Active-Templates XML-based Messaging Protocol). The Router is a 100% Java application, utilizing extensive use of threads and a layered, object-oriented architecture for efficiency.

The core of the router is a loop thread which creates a ServerSocket to listen for socket connections and, on each one, constructs a thread, implemented as the inner class MessageRouter.MessageRouterThread, to handle incoming message traffic. This loop thread can be terminated by sending in an interrupt message through the shutdown method of MessageRouter.

MessageRouterThread encapsulates the information connected to a client. It handles sending messages to the client, and receiving the messages from the client to forward to other clients or the server. The MessageRouter itself handles the details of this forwarding so that the threads aren't aware of each other. This thread also creates a "WorkerThread", the MessageForwardThread, which handles forwarding messages to clients separately, but controls synchronization, so that the other clients are not affected by significantly long downloads. The methods processString and (overloaded) processRouterAction do most of the listening work. The remaining methods handle specific requests of the Router by the client.

The Router has some built-in capabilities, usually implemented inside or alongside MessageRouterThread, such as the ability to respond to a "getAbout" message for returning versioning and status for the Router, MetaRouter, and SERVER.

A special component to the Router is the MessageServer. This is a pluggable object implementing a specific interface. Messages with the "to" field of "SERVER" will go to this component instead of to other clients in the system. The Server in the delivered ActiveTemplates system is the ContextMessageServerSocket, which handled a ServerSocket that the PlanManager or TIM communicated through, since those components were written in C++. For testing purposes, a "DummyMessageServerSocket" was written that could be used to simulate the messages from the TIM through this socket.

The MessageRouter is meant to be derived from for specializations and new features to be added. Two features added for Active Template's needs were XSL Transformations and the MetaRouter, where Routers could forward messages to clients of other routers on other machines.

Derived from MessageRouter is MetaMessageRouter, which implements router-to-router communications. The implementation overrides several methods including sendMessage and the MessageRouterThread class to handle requests for monitoring the status of the router or forwarding requests to other routers. The MetaRouter itself is just another

MessageRouter, with UniqueNameMessageServer as its server. The class, MetaMessageRouter, interacts with a class, MetaRouter, which is a client to this central router.

The standard local router may use a different SERVER than the standard MessageServer when using meta-router communications to talk to a central server. This server, called ServerToCentralServer, takes all messages meant for the local "SERVER" and forwards them unchanged to the SERVER on the meta-router's machine. This allows applications to work unchanged, whether they are dealing with a local or remote TIM.

Derived from MetaMessageRouter is TransformMetaMessageRouter, which with the help of XSLRouterHelper, handles XSL transformations. The helper class handles persistent storage of .xsl files and registrations, as well as the transforming call itself (using JAXP, so the method is portable regardless of the XSL implementation). The transformation system allows messages to be transformed at the sender's request, or be automatically transformed at the client's request when coming from a specific sender (including the SERVER). One use of this feature was in taking the results of an SQL table query in the TIM, and returning HTML to be rendered in a query client.

All the components in the Router have a single logger available, with multiple levels of reporting data, and the ability to display the debug information in a Swing-based GUI window. This is implemented as a "singleton" object, and the debug level is set by a system property at startup time. This window is shared among the entire JVM, so if a client and the router are sharing the same JVM, then they will both share the same debug window.

One final SERVER implementation that is used for testing purposes is the EchoMessageServer, which just echoes back to the sender what's sent to it.

A java-based client-side library is provided, with an abstract base implementation for making connections, sending requests, and receiving messages. A concrete subclass, Lax2MessageClient, provides an implementation that uses Lax2, a SAX-based ContentHandler that provides multicasting capabilities and uses reflection to search for method names matching signatures like "start_element", allowing content handlers to greatly simplify the methods used to handle SAX events. Two example content handlers automatically used by Lax2MessageClient are the PeerListInterpreter, which listens for changes to the list of connected peers to the client and fires JavaBean events, and the GetAboutHandler, which automatically handles the getAbout requests from other peers and the router and server, and sends its about information upon request by other peers. MessageClient (and its Lax2 derivative) have some legacy methods that make reference to a "remote" object. These can be ignored, as they refer to an older peer-to-peer based implementation of router-to-router communications.

For dependencies with COTS (well, open-source) products, the Router requires the library from the book, Building Parsers With Java by Steve Metsker (bpwj.jar), and the regexp library from the Jakarta program at Apache (jakarta-regexp-1.2.jar). The testing code requires Junit, at least version 3.6. The TransformMetaMessageRouter, the test

clients, and Lax2MessageClient require the Xerces XML Parser for Java, version 1.4.x and the transformer also needs the Xalan XSLT engine, version 2.2. The system as delivered is untested against more recent releases of Xerces, Xalan, or the JAXP standard from Sun Microsystems. It also has not been tested or approved for use with JDK versions greater than 1.3.1.

Appendix D – Plan Coordinator User’s Guide

D-1 Functionality

The Plan Coordinator is a web based applet that facilitates the creation of plans in a top down manner. The Plan Coordinator allows the user to create the framework for the overall plan and assign parts of the plan to their respective coordinators. The Plan Coordinator allows the senior coordinator to monitor the progress of the planning process, and it provides visual clues if a problem arises so that he can take action to correct it. Plan creation and monitoring is facilitated through an easy to use Graphical User Interface (GUI) so that the task of plan creation and management are easy to do.

The Plan Coordinator uses Java Web Applet Technology so that the software is easily accessible via the Internet or any private Intranet. This means that a user can access the software from any location that has access to the network where the software has been set up.

The Plan Coordinator uses the Structured Data Model (SDM) that was developed under the Active Templates program. This set of tables resides in a database and the Plan Coordinator persists all of its relevant information in these tables. This means that the Page 50 of 12 data the Plan Coordinator creates and works with is available to any application that is designed to work with the SDM, so that the basic outline of a plan can be created with the Plan Coordinator and then more specific work can be done by other applications that also use the SDM for persistent data storage.

D-2 System Requirements and Installation

The requirements for the client machine are few. The Java SDK version 1.3.1 or higher must be installed. If the Plan Coordinator is configured to use the JDBC-ODBC bridge drivers to access the database, a compatible ODBC driver must be installed. If the HTTP interface is being used, there is no need to install an ODBC driver.

On the machine where the applet is installed, Java is also required as well as an ODBC driver. This installation will most likely be handled by an administrator and is of no concern to the user.

Accessing the Plan Coordinator is done by opening up a web browser and going to the URL where the Plan Coordinator has been installed.

D-3 The Plan Coordinator

The Plan Coordinator has multiple tabs to accomplish different tasks. This section will go through each of these tabs and explain how they work and what they do.

Figure 1 shows the Plan Coordinator in its entirety. The window is divided up into two sections. The left section is a JTreeTable that shows the complete structure of the plan. Each node of the tree is a Plan Element, and any plan element may have multiple

elements underneath it. Right Clicking on the JTreeTable will bring up a popup menu that has several different capabilities listed.

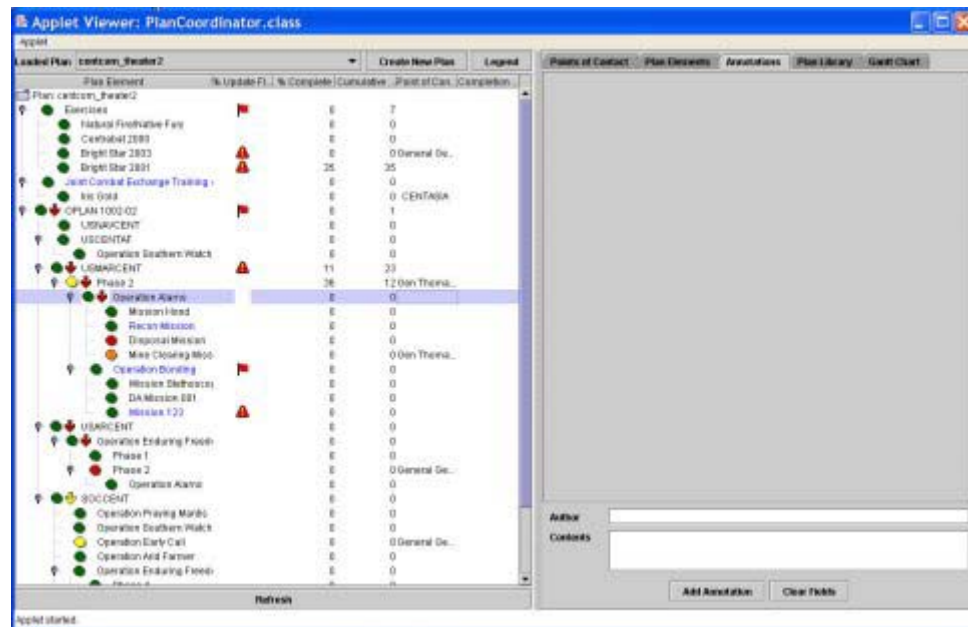


Figure D-1 - The Plan Coordinator

The right section of the Plan Coordinator window contains multiple tabs that contain different GUI's for performing different tasks. Initially the Point of Contact tab is visible.

D-3.1 Plan Element View

The Plan Element view is a JTreeTable that contains all of the elements of the plan. Each node in the view represents a row in the Plan Element table in the Structured Data Model (SDM). The view is composed of 6 columns. The first column is the name of the plan element. Each node contains a name and an icon that represents the status of that element. The circle icon shows the status of that particular element. There may also be an arrow icon displayed, if so this means that an element underneath that element has an alert status. For instance the node Disposal Mission is red indicating a problem with that element of the plan, if you look at its parent node, Operation Alamo; you will see that it has a red arrow next to its green circle. This tells you that something is wrong with this plan, and that the problem occurs somewhere below Operation Alamo. As you can see the status is rolled up such that the highest alert status will be displayed. If one element has an orange status, and another has a red status, the parent node will display a red arrow because that is the higher problem status.

Loaded Plan		centcom_theater2	Create New Plan	Legend
Plan Element	% Update Fl	% Complete	Cumulative	Point of Con
Plan: centcom_theater2				
Exercises		0	7	
Natural Fire/Native Fury		0	0	
Centrabat 2000		0	0	
Bright Star 2003		0	0	0 General Ge...
Bright Star 2001		35	35	
Joint Combat Exchange Training		0	0	
Iris Gold		0	0	0 CENTASIA
OPLAN 1002-02		0	1	
USNAVCENT		0	0	
USCENTAF		0	0	
Operation Southern Watch		0	0	
USMARCENT		11	23	
Phase 2		36	12 Gen Thoma...	
Operation Alamo		0	0	
Mission Hood		0	0	
Recon Mission		0	0	
Disposal Mission		0	0	
Mine Clearing Miss		0	0	0 Gen Thoma...
Operation Bonding		0	0	
Mission Stethoscoj		0	0	
DA Mission 001		0	0	
Mission 123		0	0	
USARCENT		0	0	
Operation Enduring Freedi		0	0	
Phase 1		0	0	
Phase 2		0	0	0 General Ge...
Operation Alamo		0	0	
SOCCENT		0	0	
Operation Praying Mantis		0	0	
Operation Southern Watch		0	0	
Operation Early Call		0	0	0 General Ge...
Operation Arid Farmer		0	0	
Operation Enduring Freedi		0	0	
Phase 1		0	0	

Figure D-2 The Plan Element View

You might also notice that some of the element names are displayed in the color blue. This is to let you know that the particular element has an annotation associated with it. If you were to look at the annotation pane you would be able to see what the annotation is.

The next column is the Percent Update Flag. This column can display 2 icons. The first is a red flag. This icon indicated that one of the child elements has had its percent complete value updated more recently than the parent. This flag lets you know that you might want to check the percent complete of the flagged field to insure that it is up to date. The exclamation point icon tells you which child was updated, and which child caused the red flag to appear on the parent element.

The next column is the Percent Complete field. This is a value between 0 and 100 that represents how complete the given element is. This value is set in the Plan Element tab.

The next column is the Cumulative Percent Complete field. How complete any given element is really depends upon how complete all of the tasks the comprise it are. Cumulative Percent Complete is a way of using that premise to estimate how complete an

element is. The Plan Coordinator looks at all of the tasks below the selected node and uses how complete they are to derive a value for the selected element. The Plan Coordinator assumes that all tasks have equals importance, which may nor may not be the case, so this value is simply a guide to assist the coordinator in assessing the situation. It's also helpful for noticing disparities in the Percent Complete field. If the cumulative percent complete says 75% and the percent complete says 5% then most likely there is a problem that needs to be looked at. This value is calculated every time the percent complete value changes for a given element.

The next column is the Point of Contact field. This is the name of the person who is in charge of that particular part of the plan. Assigning names to an element is easy. Simply click on the name of the person in the Point of Contact window (on the right section), drag it over to the plan element view, and drop it on the plan element you want to assign him to. You can also assign a point of contact via the plan element tab.

The last column is the completion date. This value is assigned in the plan element tab.

Across the top of the JTreeTable there are a few items of interest. The first is a list box that contains all of the plans that the Plan Coordinator knows about. You can change which plan you want to work with by simply selecting the plan in this list box. After that you have a button labeled "Create New Plan". If you want to create a new plan, click this button. A simple GUI (figure 3) will come up and you have to type in the name of the plan you want to create. Once you have done that the Plan Coordinator will begin sending the commands to create a new plan in the database and set everything up for you. This can take a few seconds, so wait until the dialog box appears telling you that the plan was created successfully before you continue your work.

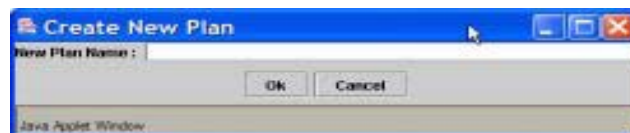


Figure D-3 Create Plan GUI

The last button up top is named "Legend". Clicking this button will display an icon legend window that describes the icons and color schemes that you see in the Plan Element Viewer.

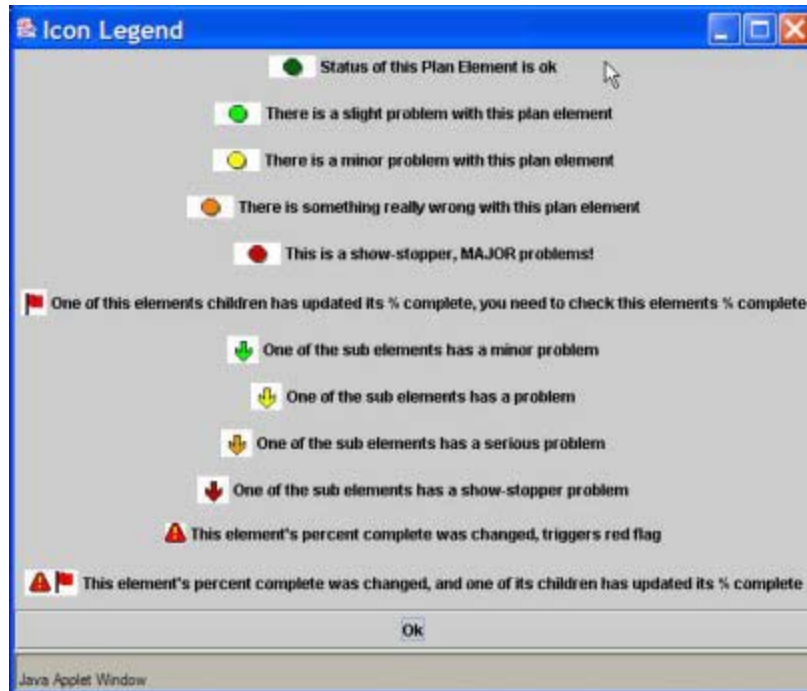


Figure D-4 The Icon Legend

At the bottom of the Plan Element Viewer you will see a button titled “Refresh”. Clicking this button will reload all of the plans from the database. So, if you suspect that some other application or user may have changed some important data you can use this button to get the latest information from the database. Be warned however, if you have made some changes to a plan element and haven’t saved them they will be lost.

D-3.2 Point of Contact Window

The point of contact window contains a table that describes the people that can be assigned to an element as the point of contact for that element of the plan. The point of contact does not have to be a person; it may be an organization, or some position of authority. For instance, you may want to have CINCPACFLT (Commander In Control of the Pacific Fleet) as the point of contact for a particular element, and you may not care exactly who that person is. So you can create an entry that represents that position regardless of what person currently holds that spot. Figure 5 shows the point of contact window.

Points of Contact		Plan Elements	Annotations	Plan Library	Garitt Chart	
Rank	First Na...	Last Na...	Organiz...	Phone	E-mail	GUID
General	George	Patton	3rd Army	678-581...	gPatton...	000-xxx-...001
Lt	Fred	Sanford	101st Inf...	xxx-xxx-...	fSanf@ar...	xxx-xxx-...002
Field Mar...	Heimede	VanRasu...	4th Shoc...	333-3443...	ehkubdd...	3333333...003
			7th Fleet	334433434	navy@na...	343434...004
			Delta For...	334433434	df@df.mil	343434...005
			CINCPAC...	334433434	df@df.mil	343434...006
			CINCLANT	334433434	df@df.mil	343434...007
Gen	Thomas	Lovecraft	CENTEUR	123-321...	tllove@a...	9nhkgdz...1f929g77...
Gen	Alex		CENTASIA			j9kutfym...

Figure D-5Point of Contact Window

To create a new entry simply press the button labeled “Add a Contact”. This will create a new row in the table that is empty of all values. Then go through the fields you wish to fill out and enter in the values. The database is updated automatically whenever you change a value in this table. The only value you should not change is the GUID field; this is used by the database as the primary key.

D-3.3 Plan Element Tab

The plan element tab is used to edit the properties of a selected plan element. To access this tab, select a plan element in the tree view and then click on the Plan Element tab on the right section. The plan element tab has different controls that allow you to change the properties of a plan element. You can change its name, its type, and many other things. You can associate a SOF file with a plan element so that when you double click on the plan element in the tree view, that SOF file will be launched in the SOFTools application. You can change the point of contact with the point of contact list box, as well as changing the times associated with the element. There is a list box that allows you to change the status for the element, as well as a slider that lets you set the percent complete for the selected element. Whenever you make a change to any of the properties you will notice the “Save Changes” button at the bottom turns red. This indicates that some change has been made and hasn’t been saved. The changes will not go into effect until you press this button.

The “Restore” button allows you to restore the properties to the values they contained at the last save. So if you make several changes, and haven’t saved them, you can hit restore to undo all of your changes. Once you hit “Save Changes” the values are permanent however.

The “Add as Child” and “Add as Sibling” buttons are for creating a new plan element. Just fill in the values that you want your new element to have and then select the appropriate button. “Add as Child” will make your new element a child of the selected element. “Add as Sibling” will make your new element appear on the same level as the selected element. In this way you can create new elements quickly by modeling them after existing elements.

The screenshot shows a software window with five tabs: "Points of Contact", "Plan Elements", "Annotations", "Plan Library", and "Gantt Chart". The "Plan Elements" tab is active. It contains the following fields and controls:

- Name:** A text box containing "Operation Southern Watch".
- Type:** A dropdown menu with "MISSION" selected.
- Point of Contact:** A dropdown menu with "None" selected.
- Start Time (Click to Change):** A text box containing "HStart".
- End Time (Click to Change):** A text box containing "HEnd".
- Completion Time (Click to Change):** A text box containing "NONE".
- SOF File Name:** A text box containing "example1.sof" and a "browse" button.
- Status:** A dropdown menu with "Dark Green - Everything Ok" selected.
- Percent Complete:** A progress bar with a slider handle at 0% and a scale from 0 to 100.

At the bottom of the window are five buttons: "Add as Child", "Add as Sibling", "Save Changes", "Restore", and "Delete".

Figure D-6 - The Plan Element Tab

D-3.4 Annotations Window

The annotation window allows you to add annotations to a particular plan element. If a plan element has annotations associated with it, the text color of the name of the element in the tree view will be blue. To add an annotation, just fill in both text boxes and press “Add Annotation”. Figure D-7 shows the annotation window.



Figure D-7 Annotation Window

D-3.5 Plan Library

The plan library is a place where all of the plans and the elements they contain can be accessed without changing the plan that you are currently working on. Elements or entire trees of elements can be dragged from the plan library and dropped into your current plan. Think of the plan library as a good place to store templates of plans. Suppose I have an operation in plan A that I want to add to plan B. Lets say the operation is quite involved and contains lots of elements. Entering that operation into plan B by hand is time consuming and unnecessary since I already have it defined in another plan. So I go to the plan library and select Plan A. I drag the operation over to plan B, which is my current plan in the tree view, and drop it where I want it to be added. It's as simple as that. The plan library is also the only place where you can delete a plan.

The “Reload Plan” button will take the currently selected plan, in the plan library, and reload its data from the database. The “Reload List” button will reload the available plans list from the database.

The plan library can be used to store templates. This allows you to create a database that holds templates of plans. When you are working on a new plan you can go to this template database, via the plan library, and drag and drop plan elements into your plan so that you don't have to recreate elements every time you start a new plan.

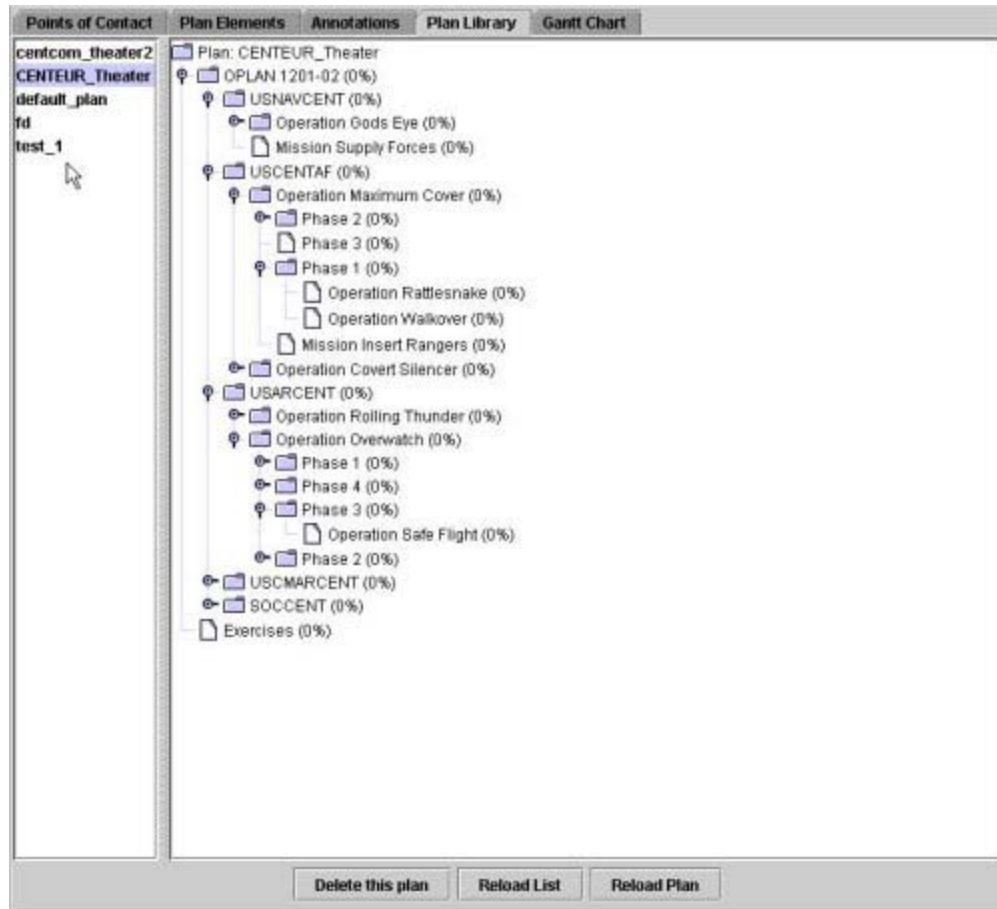


Figure D-8 - The Plan Library

D-3.6 Gantt Chart

The Gantt Chart is a window that gives you information about the timeframes in which plans exist or operate. You can't actually change any of the times via the Gantt chart, but you can look at the overall operational time in which a plan takes place, as well as the timing between various elements of a plan.

The Gantt window is divided into two panels. The left panel is a tree that contains all of the elements of the plan. The right panel shows colored bars denoting the duration of the plan. You can click on the dates in the right panel to make it zoom down to give you a more detailed view of the timing of a plan.

Clicking the "Reload" button will cause the selected plan to be reloaded from the database to insure that you have the latest data available.

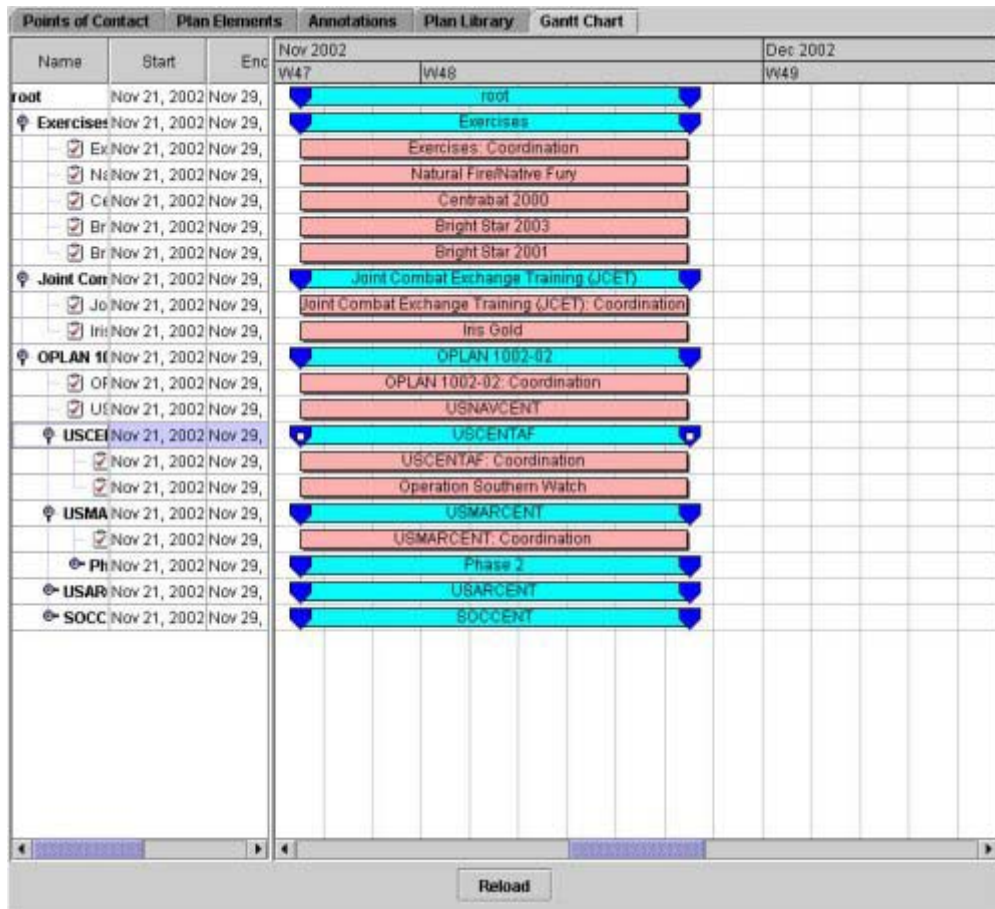


Figure D-9 The Gantt Chart

D-4 Plan Coordinator Configuration

There is very little configuration needed for the Plan Coordinator. There are a couple of properties in the index.html file that are worth mentioning.

```
<applet code="PlanCoordinator.class"
  archive="lax2.jar, xerces.jar, classes12.zip,
  mysqlConn.jar, isxjws-1.1r4.jar, xbt.jar, gantt-
  5.0.jar,jviews-5.0.jar"
  width="1000"
  height="500">
<PARAM NAME=DBType VALUE="MySQL">
<PARAM NAME=CENTRAL_SCHEMA VALUE="central_act_schema">
<PARAM NAME=DB_HOST VALUE="127.0.0.1">
</applet>
```

The above snippet is the applet tag in the index.html file. You notice that it contains 3 PARAM tags. The first is DB_TYPE. This describes the type of database that the Plan Coordinator will be interacting with. At the time of this writing the only two supported

DBMS's are MySQL and Oracle. The second parameter is the CENTRAL_SCHEMA. This is the name of the space where the table PLANNNAMES is stored as well as other information that isn't specific to any one plan. This is where the names of the plans are stored. The third parameter is DB_HOST. This is the IP Address of the machine where the database is installed. If that machine is the same machine that the Plan Coordinator is installed on, 127.0.0.1 or localhost are acceptable values.